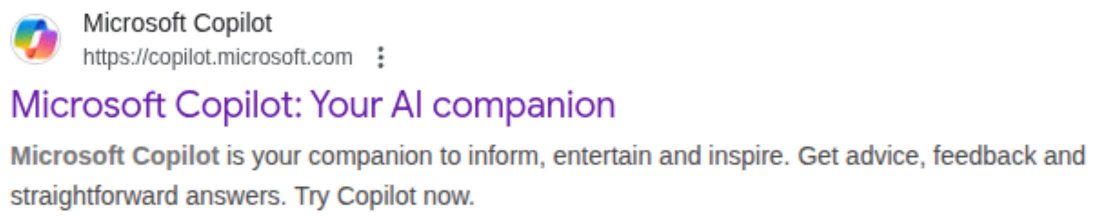
glibc FILE IO

20251025 Sat

Copyright (c) 2025 Young W. Lim.
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

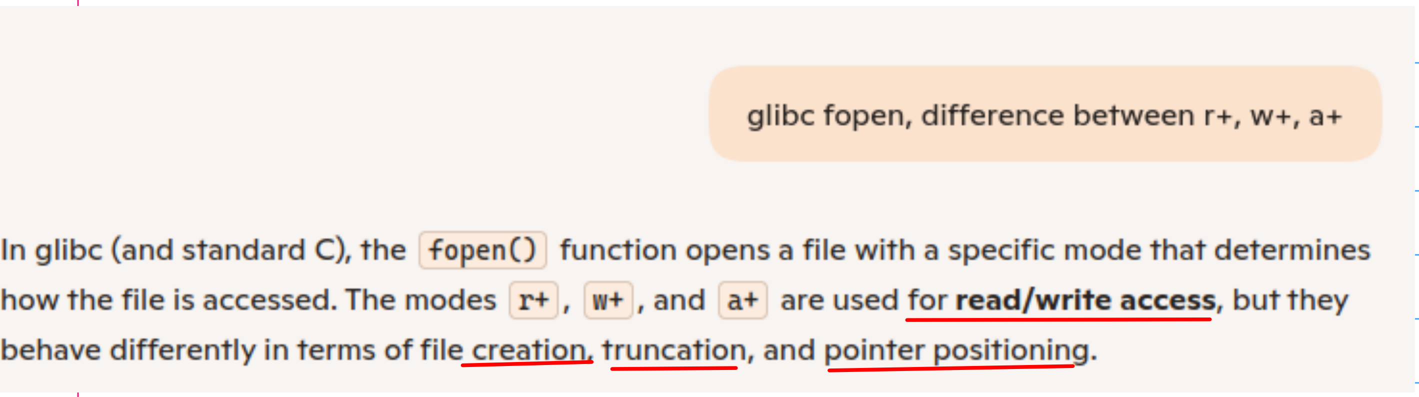
I used Microsoft Copilot to prepare this document



Microsoft Copilot
https://copilot.microsoft.com

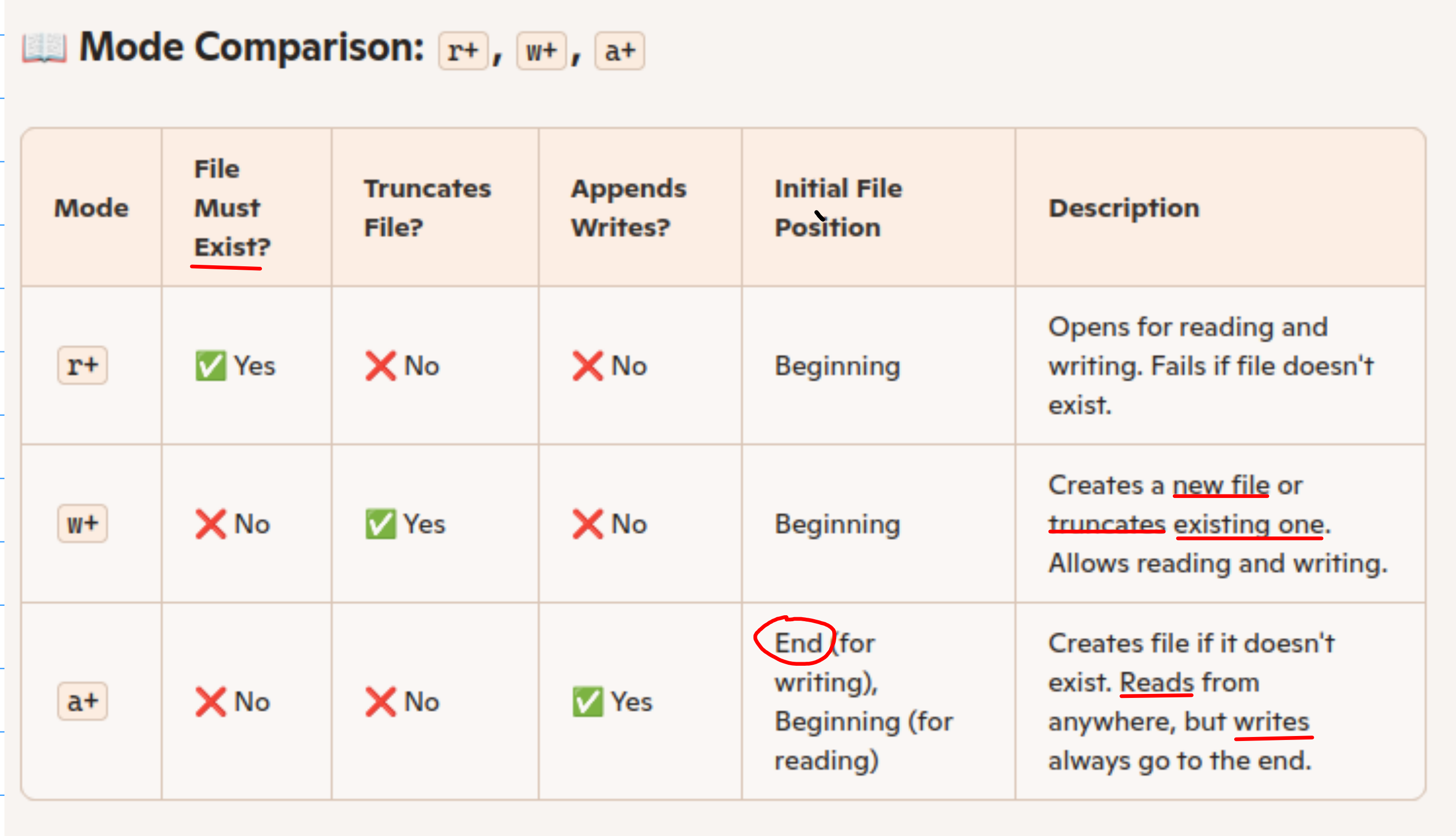
Microsoft Copilot: Your AI companion

Microsoft Copilot is your companion to inform, entertain and inspire. Get advice, feedback and straightforward answers. Try Copilot now.



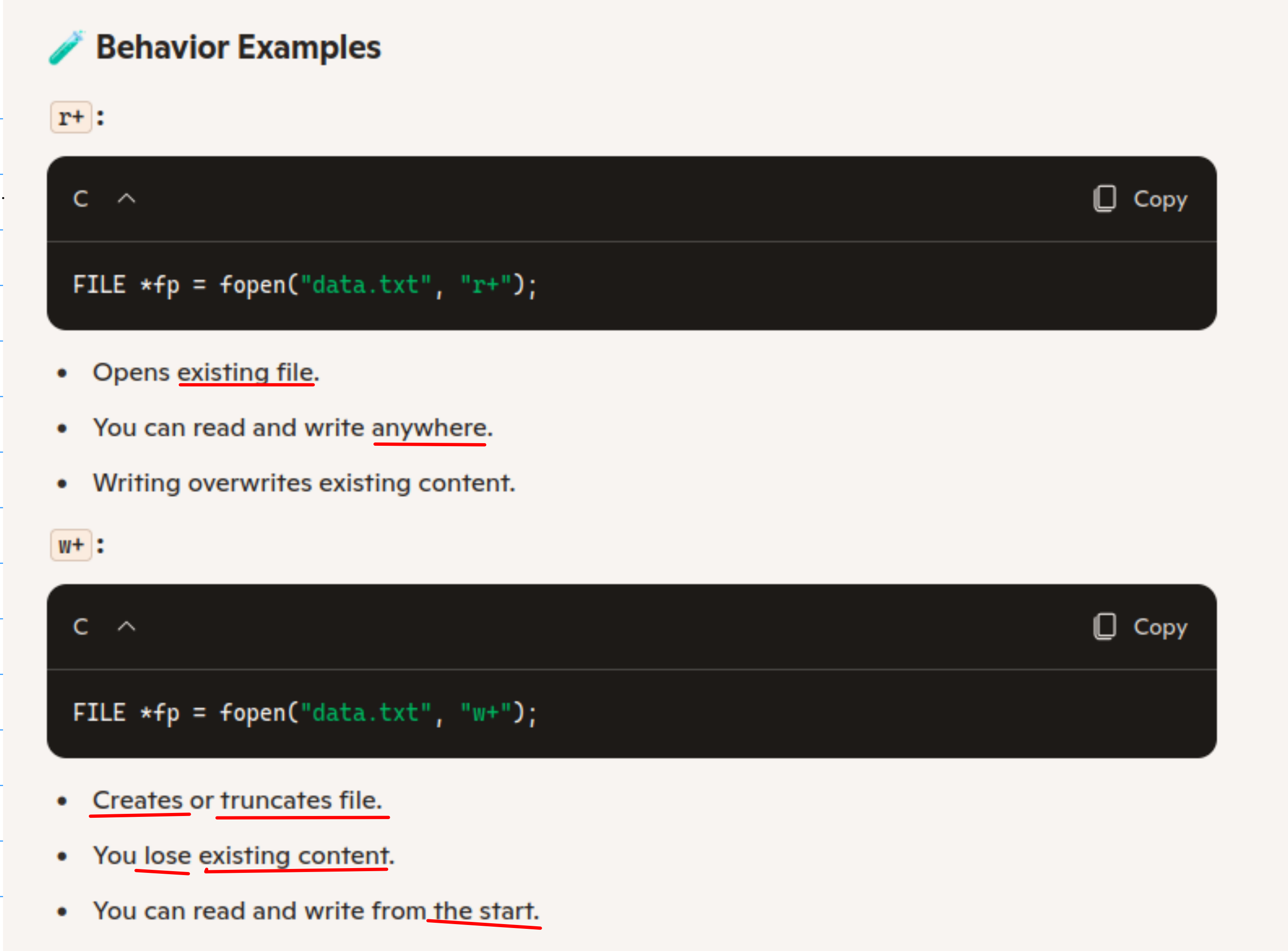
glibc fopen, difference between r+, w+, a+

In glibc (and standard C), the `fopen()` function opens a file with a specific mode that determines how the file is accessed. The modes `r+`, `w+`, and `a+` are used for **read/write access**, but they behave differently in terms of file creation, truncation, and pointer positioning.



📖 Mode Comparison: `r+`, `w+`, `a+`

Mode	File Must Exist?	Truncates File?	Appends Writes?	Initial File Position	Description
`r+`	✅ Yes	❌ No	❌ No	Beginning	Opens for reading and writing. Fails if file doesn't exist.
`w+`	❌ No	✅ Yes	❌ No	Beginning	Creates a new file or truncates existing one. Allows reading and writing.
`a+`	❌ No	❌ No	✅ Yes	End (for writing), Beginning (for reading)	Creates file if it doesn't exist. Reads from anywhere, but writes always go to the end.



🧪 Behavior Examples

`r+`:

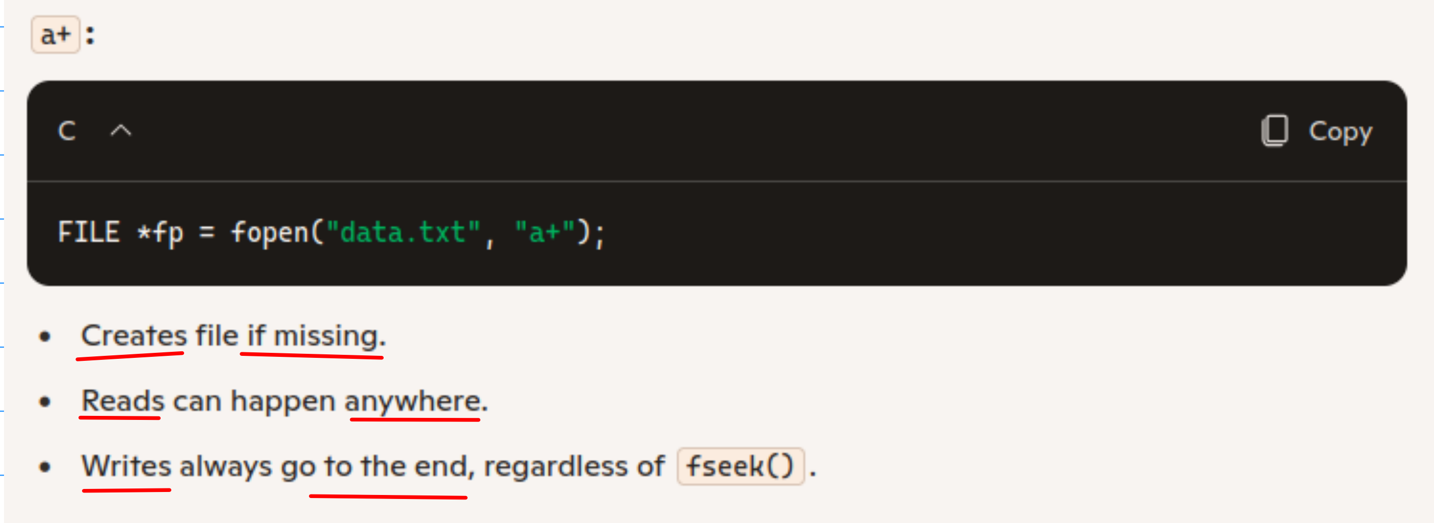
```c
FILE *fp = fopen("data.txt", "r+");
```

- Opens existing file.
- You can read and write anywhere.
- Writing overwrites existing content.

`w+`:

```c
FILE *fp = fopen("data.txt", "w+");
```

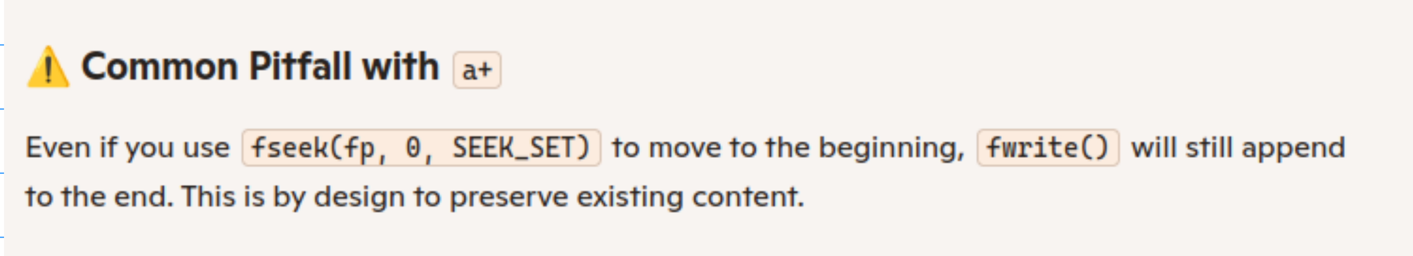
- Creates or truncates file.
- You lose existing content.
- You can read and write from the start.



`a+`:

```c
FILE *fp = fopen("data.txt", "a+");
```

- Creates file if missing.
- Reads can happen anywhere.
- Writes always go to the end, regardless of `fseek()`.



⚠️ Common Pitfall with `a+`

Even if you use `fseek(fp, 0, SEEK_SET)` to move to the beginning, `fwrite()` will still append to the end. This is by design to preserve existing content.

glibc fopen

In glibc, `fopen()` is a standard C library function that opens a file and returns a pointer to a `FILE` stream, enabling buffered I/O operations. It wraps low-level system calls like `open()` and manages internal buffers for efficiency.

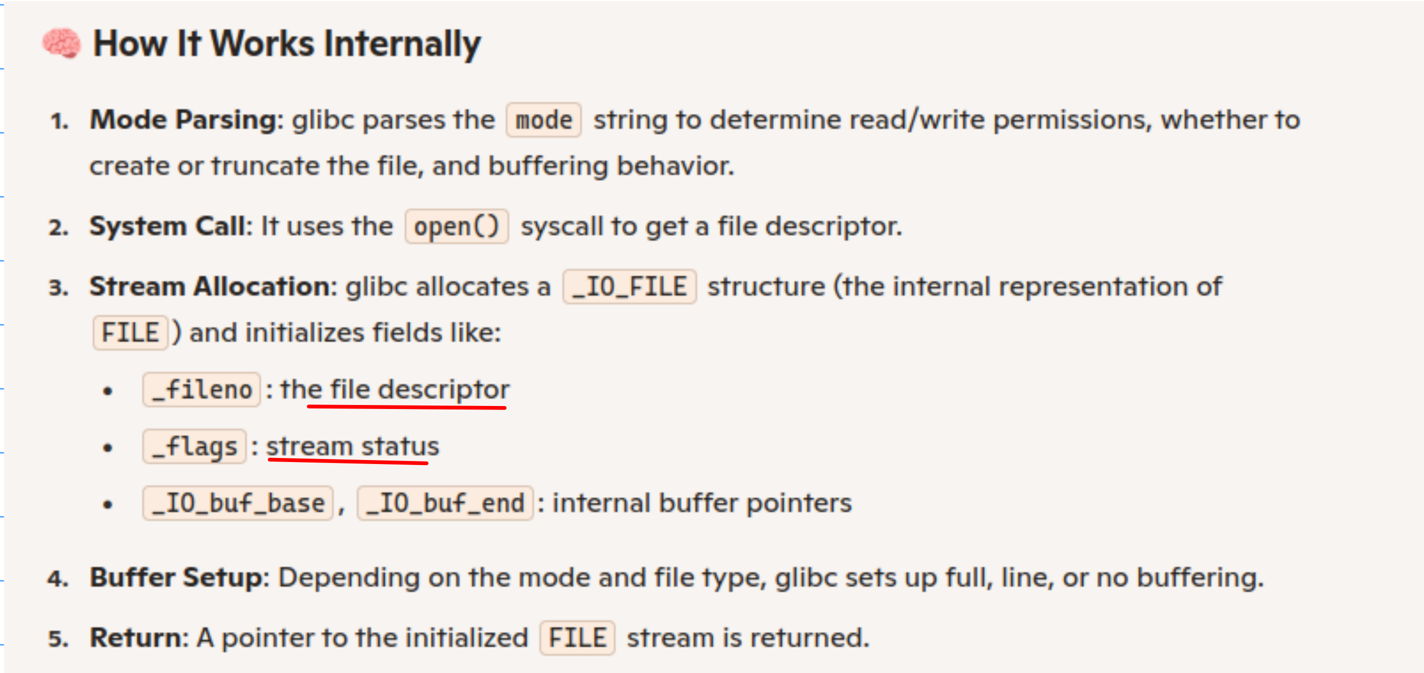
📖 `fopen()` in glibc: Overview

The `fopen()` function is defined in `<stdio.h>` and implemented in glibc as part of its standard I/O library. It creates a `FILE` stream associated with a file descriptor and sets up internal buffering and stream state.

🔧 Function Signature

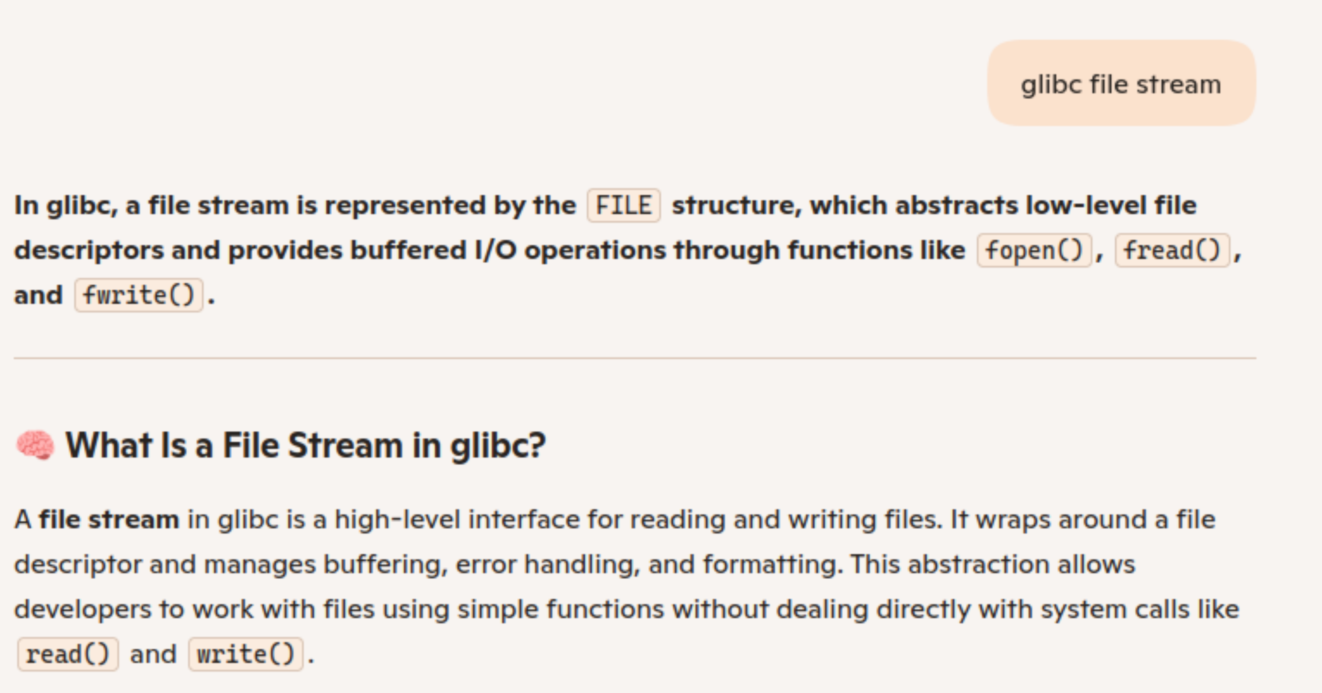
```c
FILE *fopen(const char *pathname, const char *mode);
```

- `pathname`: Path to the file.
- `mode`: A string indicating how the file should be opened (e.g., `"r"`, `"w+"`, `"a"`).



🧠 How It Works Internally

1. **Mode Parsing**: glibc parses the `mode` string to determine read/write permissions, whether to create or truncate the file, and buffering behavior.
2. **System Call**: It uses the `open()` syscall to get a file descriptor.
3. **Stream Allocation**: glibc allocates a `_IO_FILE` structure (the internal representation of `FILE`) and initializes fields like:
   - `_fileno`: the file descriptor
   - `_flags`: stream status
   - `_IO_buf_base`, `_IO_buf_end`: internal buffer pointers
4. **Buffer Setup**: Depending on the mode and file type, glibc sets up full, line, or no buffering.
5. **Return**: A pointer to the initialized `FILE` stream is returned.

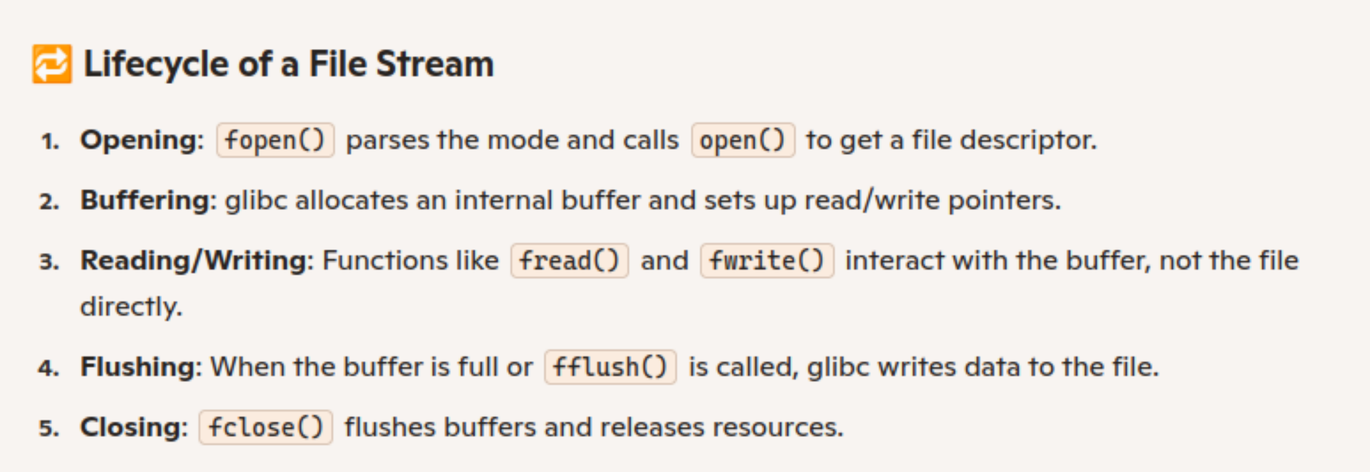


glibc file stream

In glibc, a file stream is represented by the `FILE` structure, which abstracts low-level file descriptors and provides buffered I/O operations through functions like `fopen()`, `fread()`, and `fwrite()`.

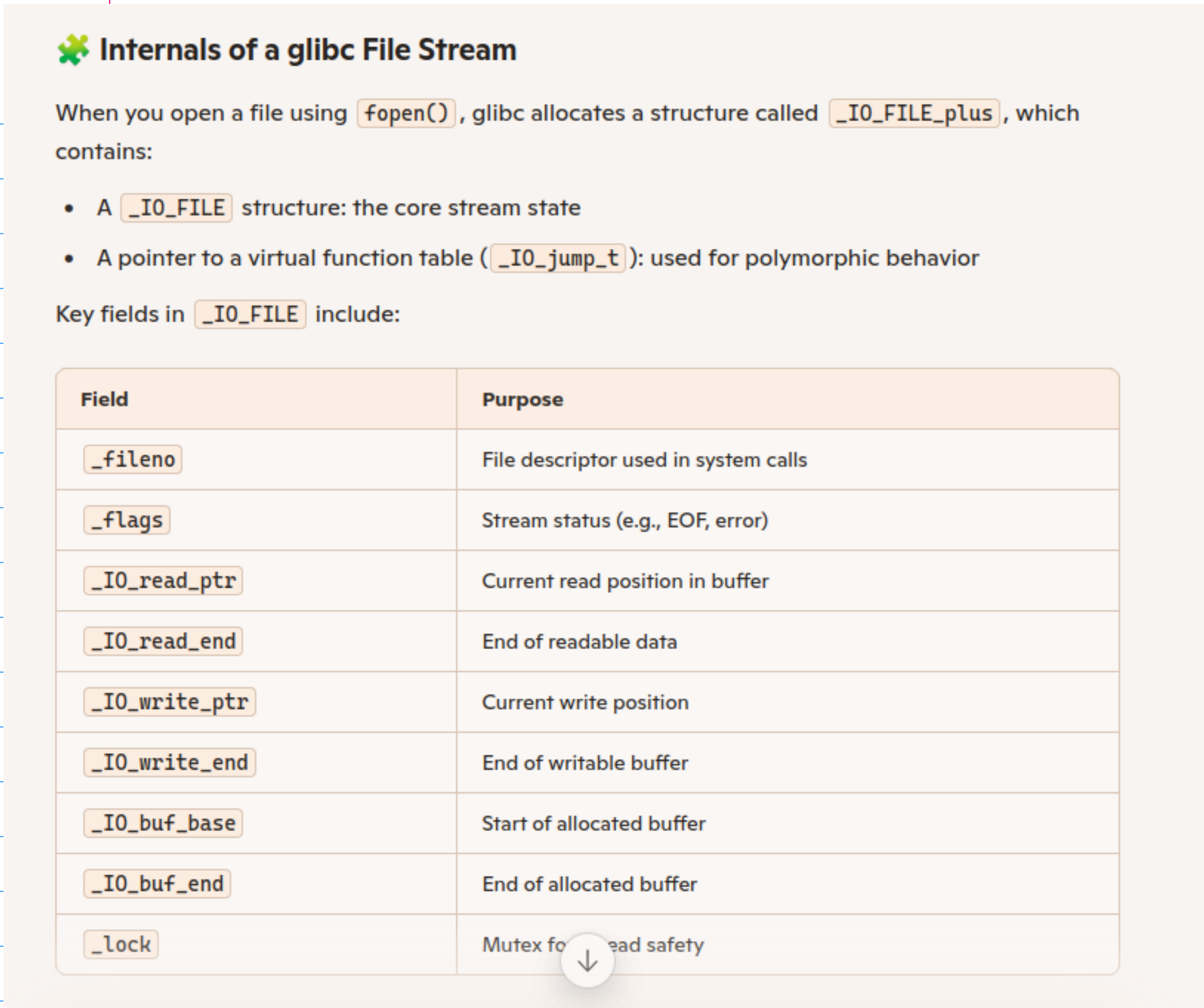
🧠 What Is a File Stream in glibc?

A **file stream** in glibc is a high-level interface for reading and writing files. It wraps around a file descriptor and manages buffering, error handling, and formatting. This abstraction allows developers to work with files using simple functions without dealing directly with system calls like `read()` and `write()`.



🔁 Lifecycle of a File Stream

1. **Opening**: `fopen()` parses the mode and calls `open()` to get a file descriptor.
2. **Buffering**: glibc allocates an internal buffer and sets up read/write pointers.
3. **Reading/Writing**: Functions like `fread()` and `fwrite()` interact with the buffer, not the file directly.
4. **Flushing**: When the buffer is full or `fflush()` is called, glibc writes data to the file.
5. **Closing**: `fclose()` flushes buffers and releases resources.



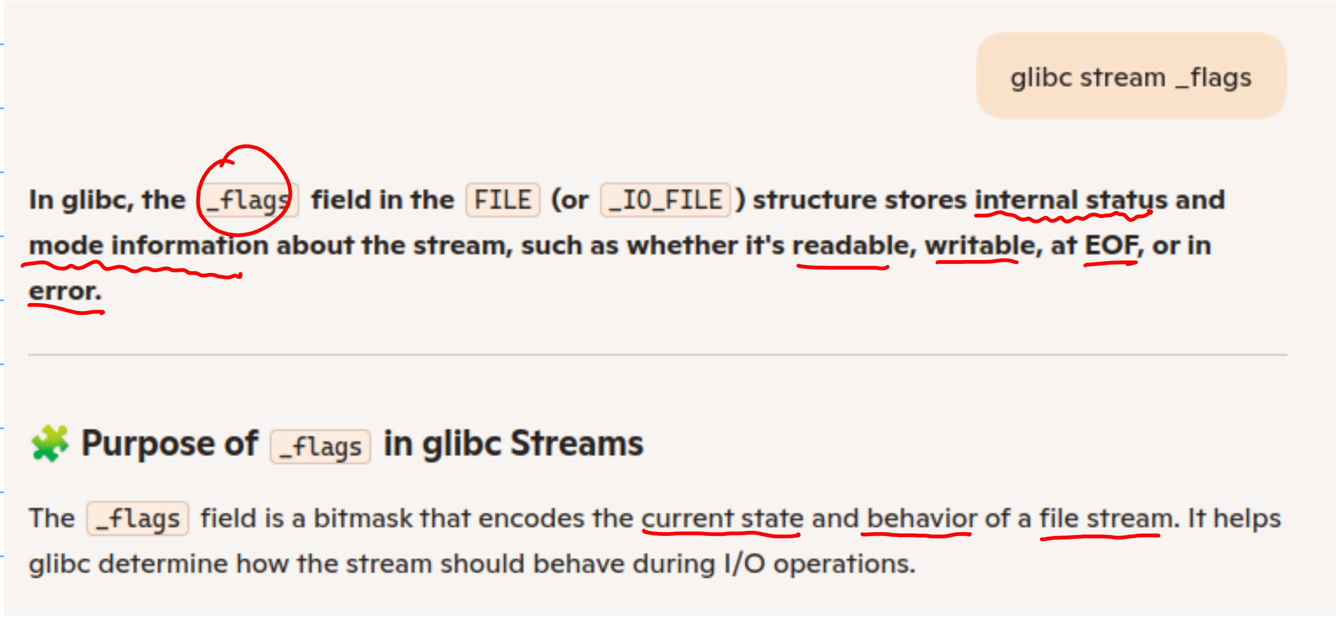
🧩 Internals of a glibc File Stream

When you open a file using `fopen()`, glibc allocates a structure called `_IO_FILE_plus`, which contains:

- A `_IO_FILE` structure: the core stream state
- A pointer to a virtual function table (`_IO_jump_t`): used for polymorphic behavior

Key fields in `_IO_FILE` include:

Field	Purpose
`_fileno`	File descriptor used in system calls
`_flags`	Stream status (e.g., EOF, error)
`_IO_read_ptr`	Current read position in buffer
`_IO_read_end`	End of readable data
`_IO_write_ptr`	Current write position
`_IO_write_end`	End of writable buffer
`_IO_buf_base`	Start of allocated buffer
`_IO_buf_end`	End of allocated buffer
`_lock`	Mutex fo[illegible]ead safety

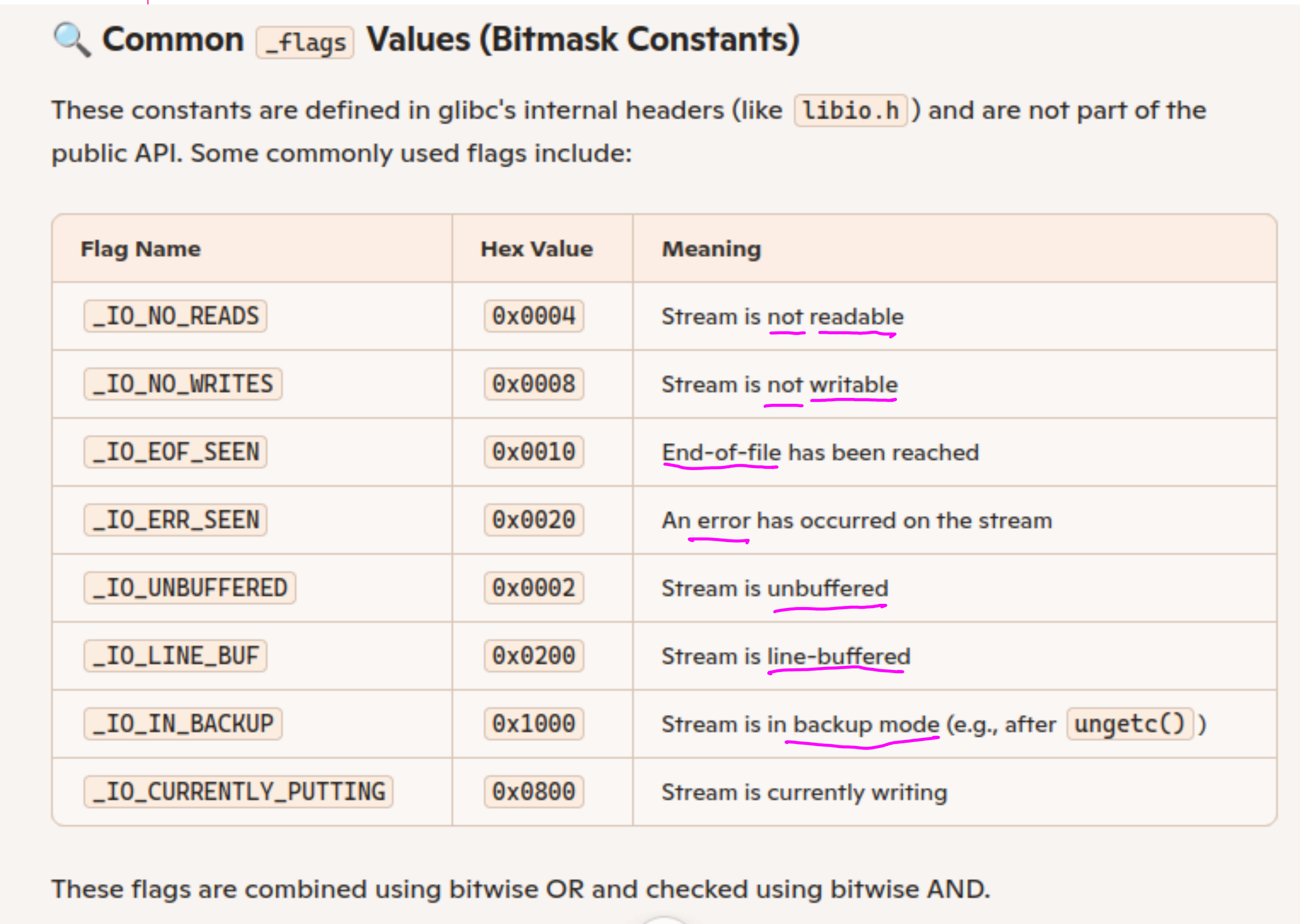


glibc stream _flags

In glibc, the `_flags` field in the `FILE` (or `_IO_FILE`) structure stores internal status and mode information about the stream, such as whether it's readable, writable, at EOF, or in error.

🧩 Purpose of `_flags` in glibc Streams

The `_flags` field is a bitmask that encodes the current state and behavior of a file stream. It helps glibc determine how the stream should behave during I/O operations.

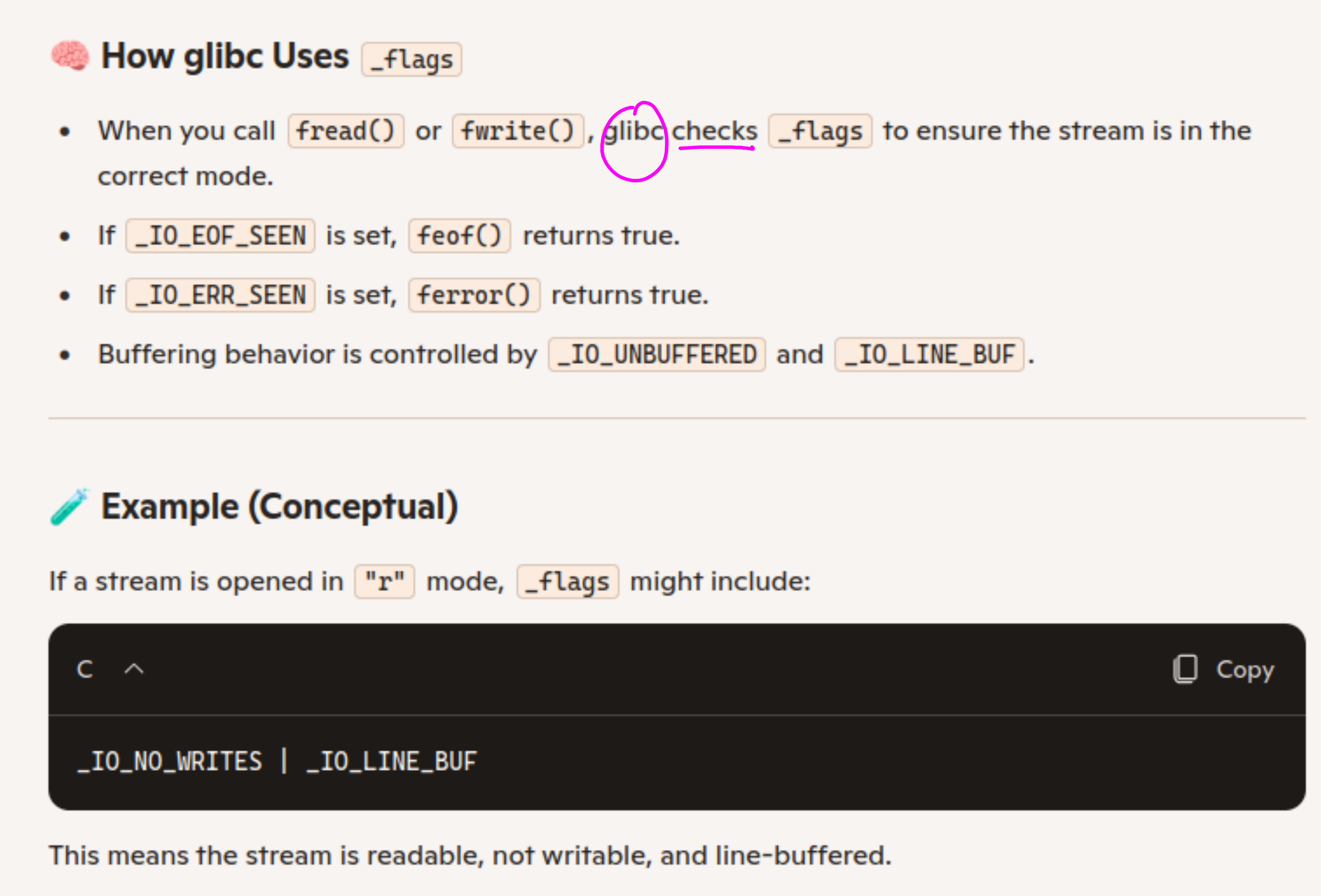


🔍 Common `_flags` Values (Bitmask Constants)

These constants are defined in glibc's internal headers (like `libio.h`) and are not part of the public API. Some commonly used flags include:

Flag Name	Hex Value	Meaning
`_IO_NO_READS`	`0x0004`	Stream is not readable
`_IO_NO_WRITES`	`0x0008`	Stream is not writable
`_IO_EOF_SEEN`	`0x0010`	End-of-file has been reached
`_IO_ERR_SEEN`	`0x0020`	An error has occurred on the stream
`_IO_UNBUFFERED`	`0x0002`	Stream is unbuffered
`_IO_LINE_BUF`	`0x0200`	Stream is line-buffered
`_IO_IN_BACKUP`	`0x1000`	Stream is in backup mode (e.g., after `ungetc()`)
`_IO_CURRENTLY_PUTTING`	`0x0800`	Stream is currently writing

These flags are combined using bitwise OR and checked using bitwise AND.



🧠 How glibc Uses `_flags`

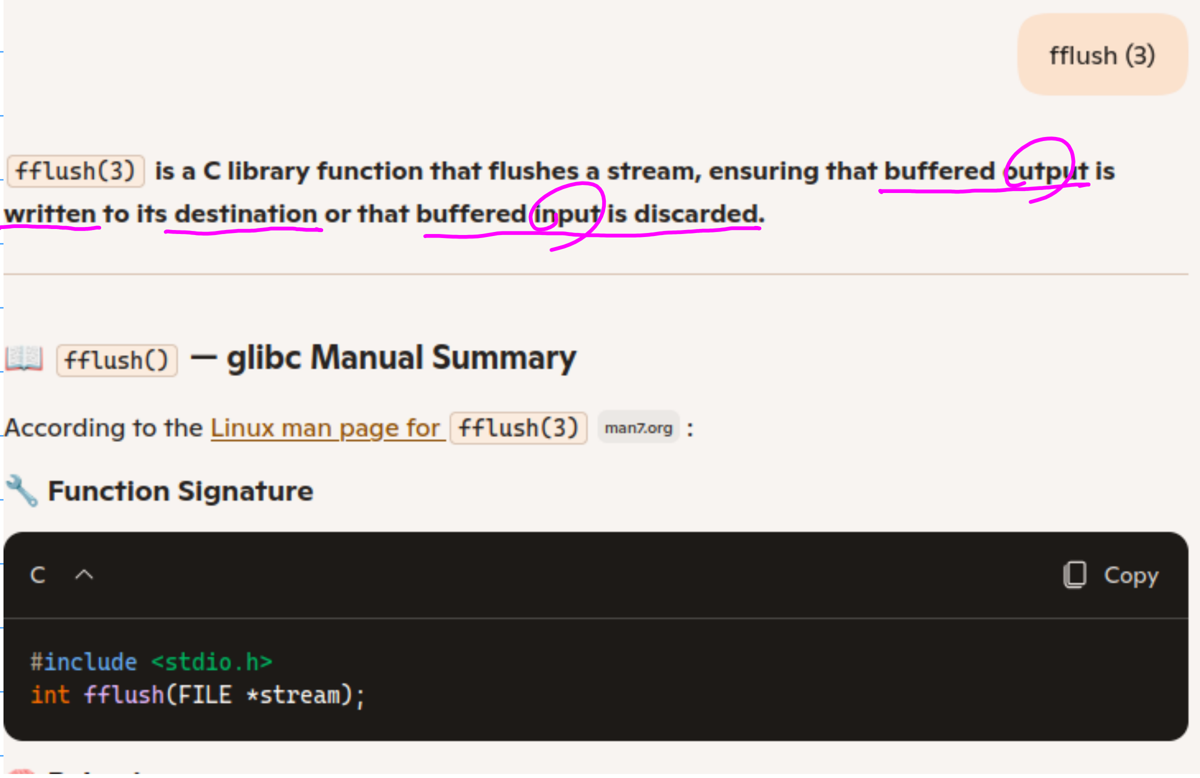
- When you call `fread()` or `fwrite()`, glibc checks `_flags` to ensure the stream is in the correct mode.
- If `_IO_EOF_SEEN` is set, `feof()` returns true.
- If `_IO_ERR_SEEN` is set, `ferror()` returns true.
- Buffering behavior is controlled by `_IO_UNBUFFERED` and `_IO_LINE_BUF`.

🧪 Example (Conceptual)

If a stream is opened in `"r"` mode, `_flags` might include:

```c
_IO_NO_WRITES | _IO_LINE_BUF
```

This means the stream is readable, not writable, and line-buffered.



fflush (3)

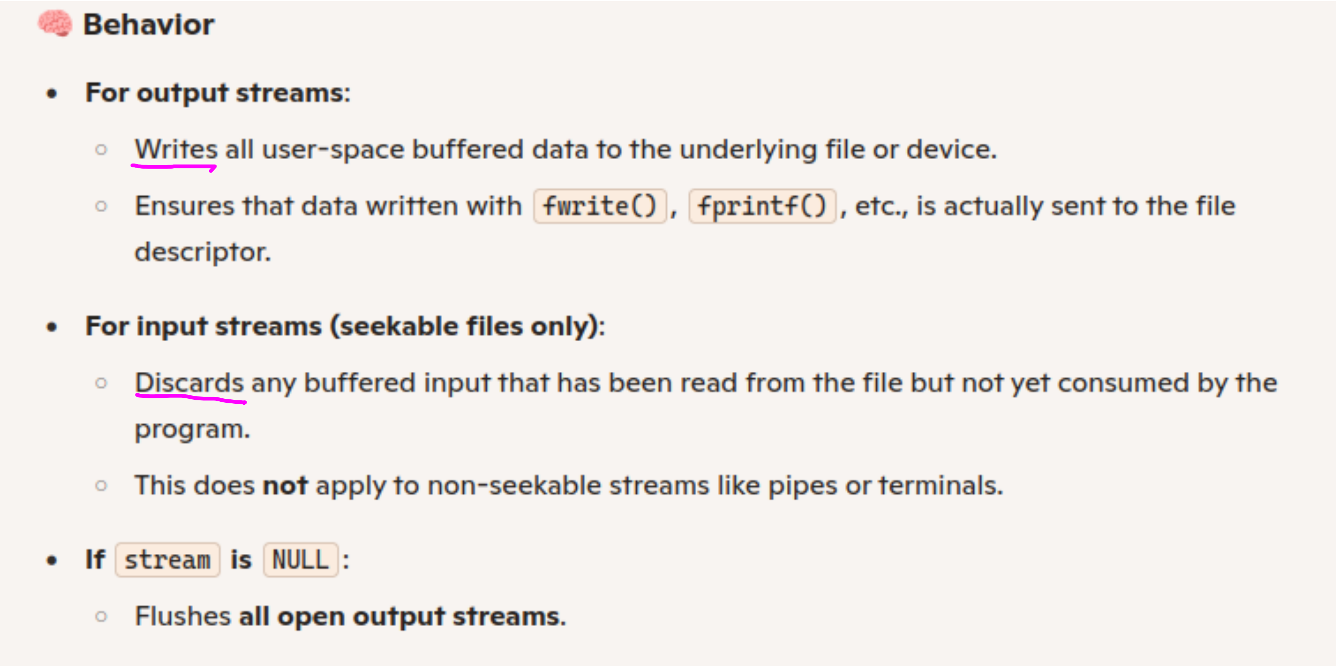
`fflush(3)` **is a C library function that flushes a stream, ensuring that buffered output is written to its destination or that buffered input is discarded.**

📖 `fflush()` — glibc Manual Summary

According to the Linux man page for `fflush(3)` man7.org :

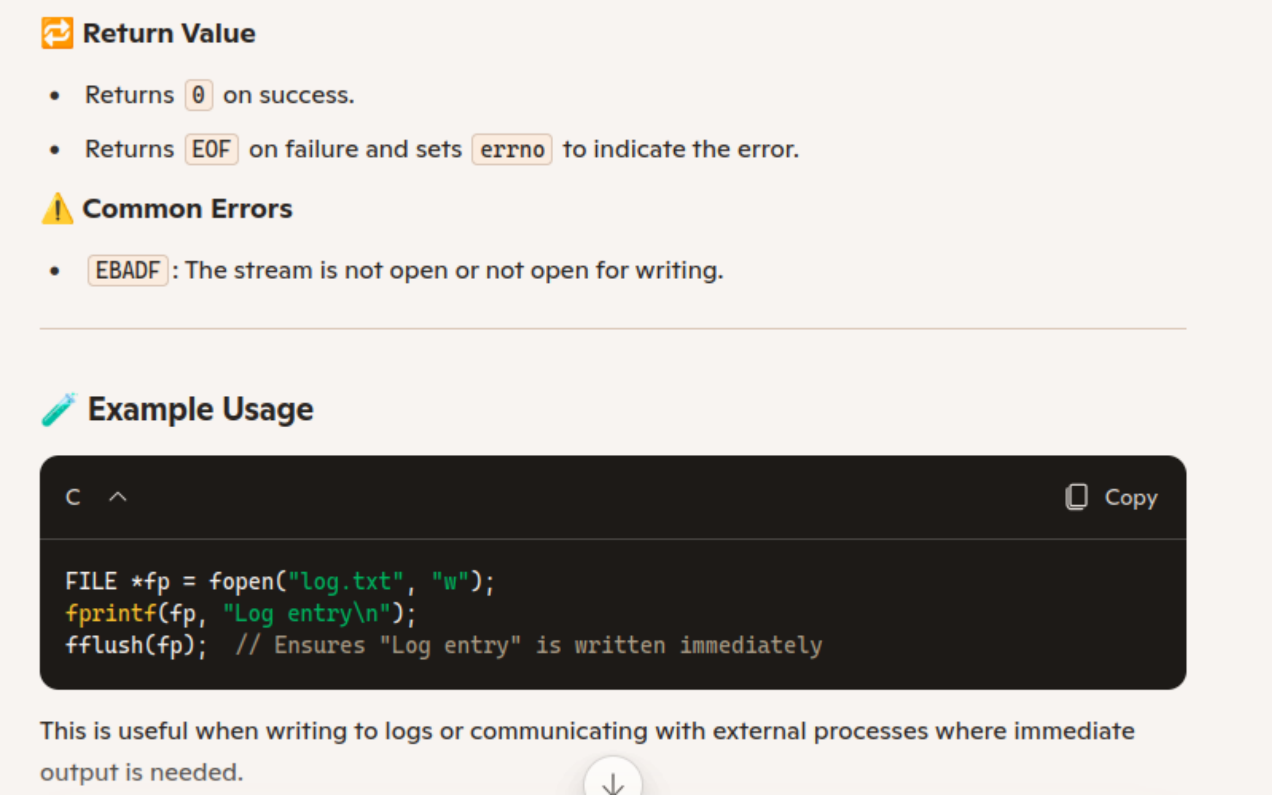
🔧 Function Signature

```c
#include <stdio.h>
int fflush(FILE *stream);
```



🧠 Behavior

- **For output streams:**
  - Writes all user-space buffered data to the underlying file or device.
  - Ensures that data written with `fwrite()`, `fprintf()`, etc., is actually sent to the file descriptor.
- **For input streams (seekable files only):**
  - Discards any buffered input that has been read from the file but not yet consumed by the program.
  - This does **not** apply to non-seekable streams like pipes or terminals.
- **If `stream` is `NULL`:**
  - Flushes **all open output streams.**



🔁 Return Value

- Returns `0` on success.
- Returns `EOF` on failure and sets `errno` to indicate the error.

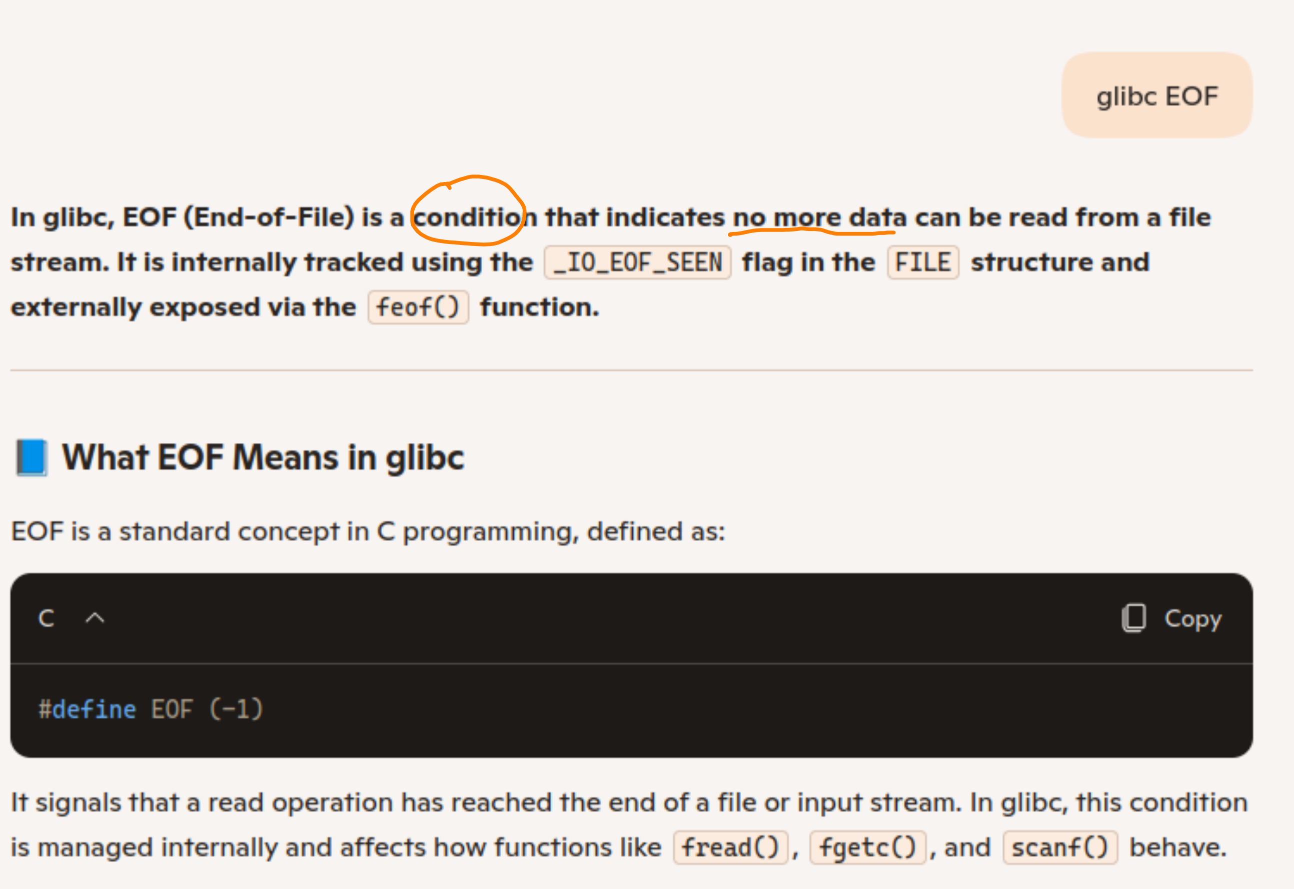
⚠️ Common Errors

- `EBADF`: The stream is not open or not open for writing.

🧪 Example Usage

```c
FILE *fp = fopen("log.txt", "w");
fprintf(fp, "Log entry\n");
fflush(fp);  // Ensures "Log entry" is written immediately
```

This is useful when writing to logs or communicating with external processes where immediate output is needed.



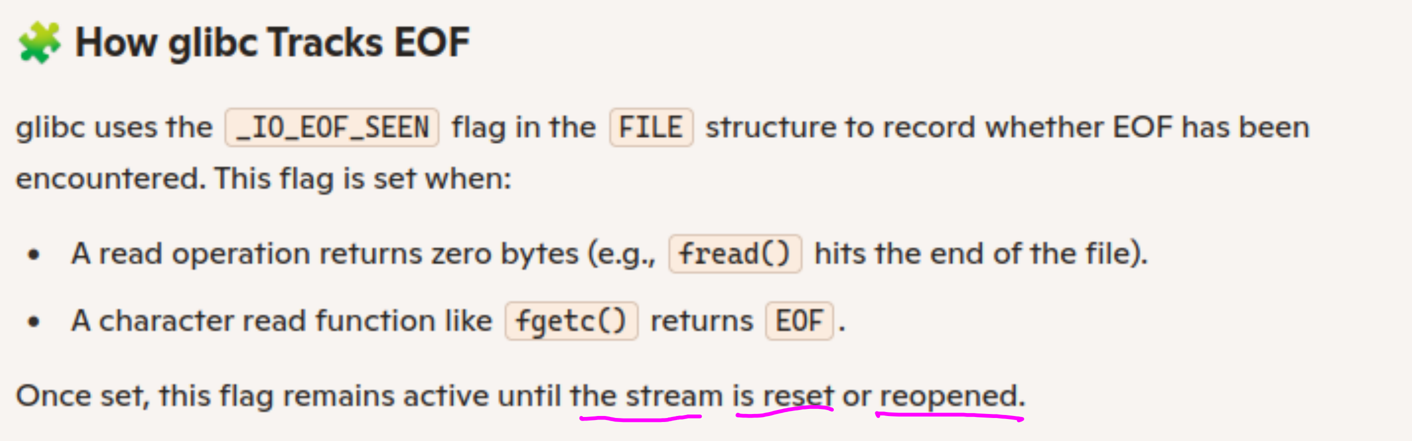
In glibc, EOF (End-of-File) is a condition that indicates no more data can be read from a file stream. It is internally tracked using the `_IO_EOF_SEEN` flag in the `FILE` structure and externally exposed via the `feof()` function.

📘 What EOF Means in glibc

EOF is a standard concept in C programming, defined as:

```c
#define EOF (-1)
```

It signals that a read operation has reached the end of a file or input stream. In glibc, this condition is managed internally and affects how functions like `fread()`, `fgetc()`, and `scanf()` behave.

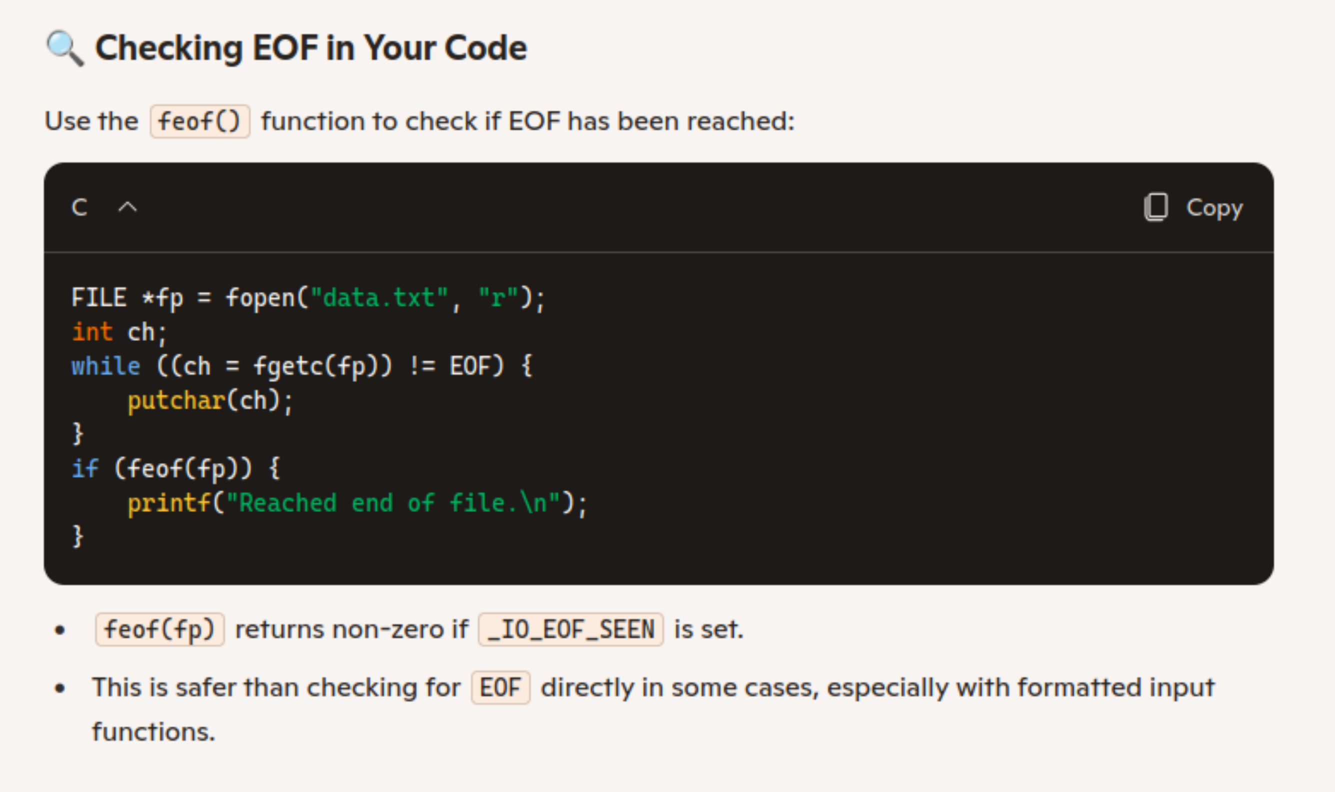


🧩 How glibc Tracks EOF

glibc uses the `_IO_EOF_SEEN` flag in the `FILE` structure to record whether EOF has been encountered. This flag is set when:

- A read operation returns zero bytes (e.g., `fread()` hits the end of the file).
- A character read function like `fgetc()` returns `EOF`.

Once set, this flag remains active until the stream is reset or reopened.

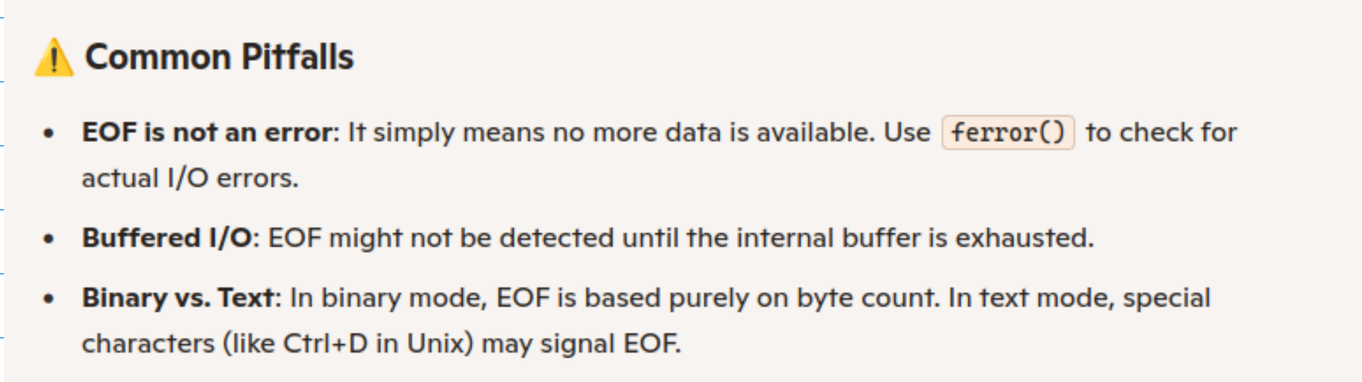


🔍 Checking EOF in Your Code

Use the `feof()` function to check if EOF has been reached:

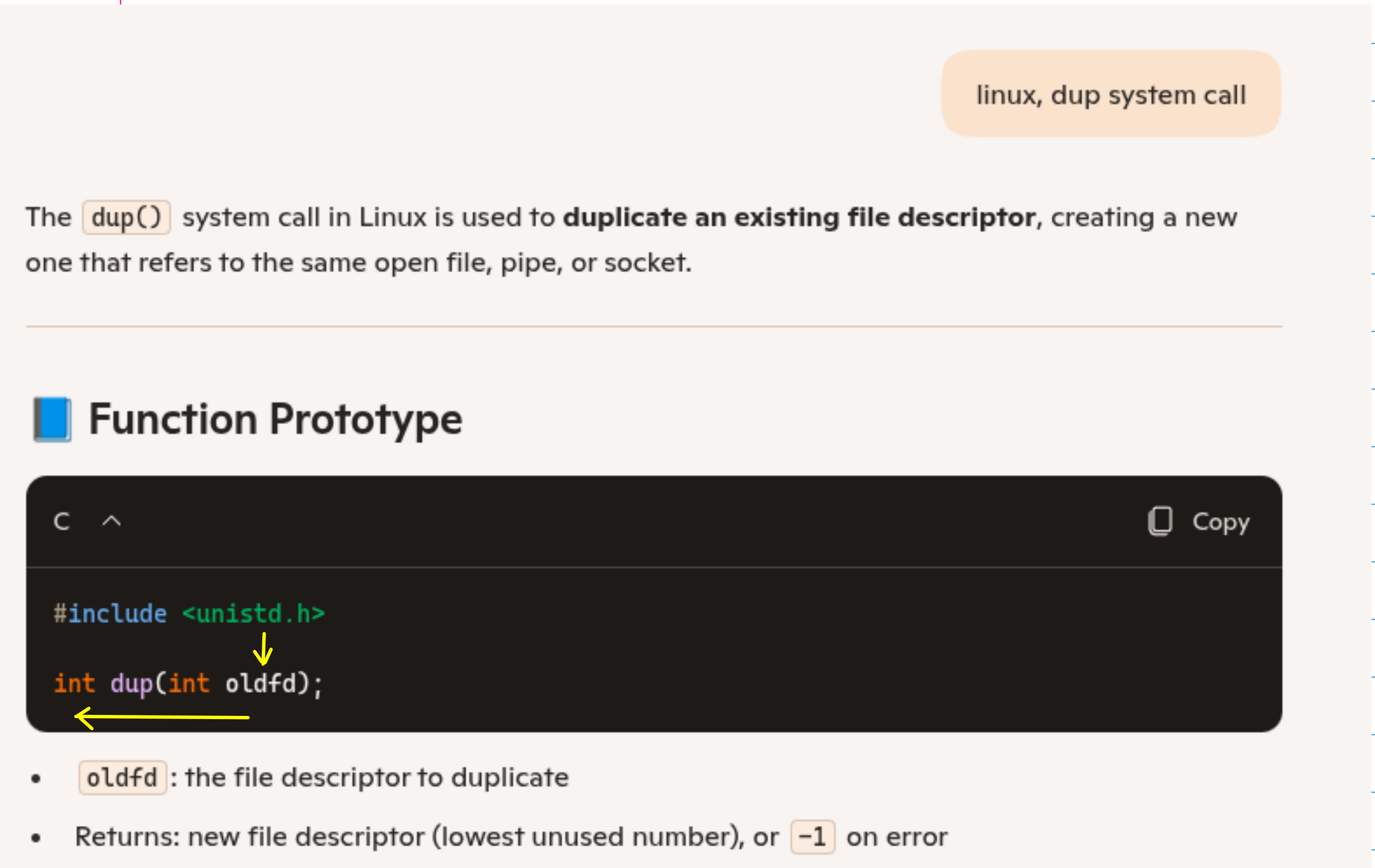
```c
FILE *fp = fopen("data.txt", "r");
int ch;
while ((ch = fgetc(fp)) != EOF) {
    putchar(ch);
}
if (feof(fp)) {
    printf("Reached end of file.\n");
}
```

- `feof(fp)` returns non-zero if `_IO_EOF_SEEN` is set.
- This is safer than checking for `EOF` directly in some cases, especially with formatted input functions.



⚠️ Common Pitfalls

- **EOF is not an error**: It simply means no more data is available. Use `ferror()` to check for actual I/O errors.
- **Buffered I/O**: EOF might not be detected until the internal buffer is exhausted.
- **Binary vs. Text**: In binary mode, EOF is based purely on byte count. In text mode, special characters (like Ctrl+D in Unix) may signal EOF.



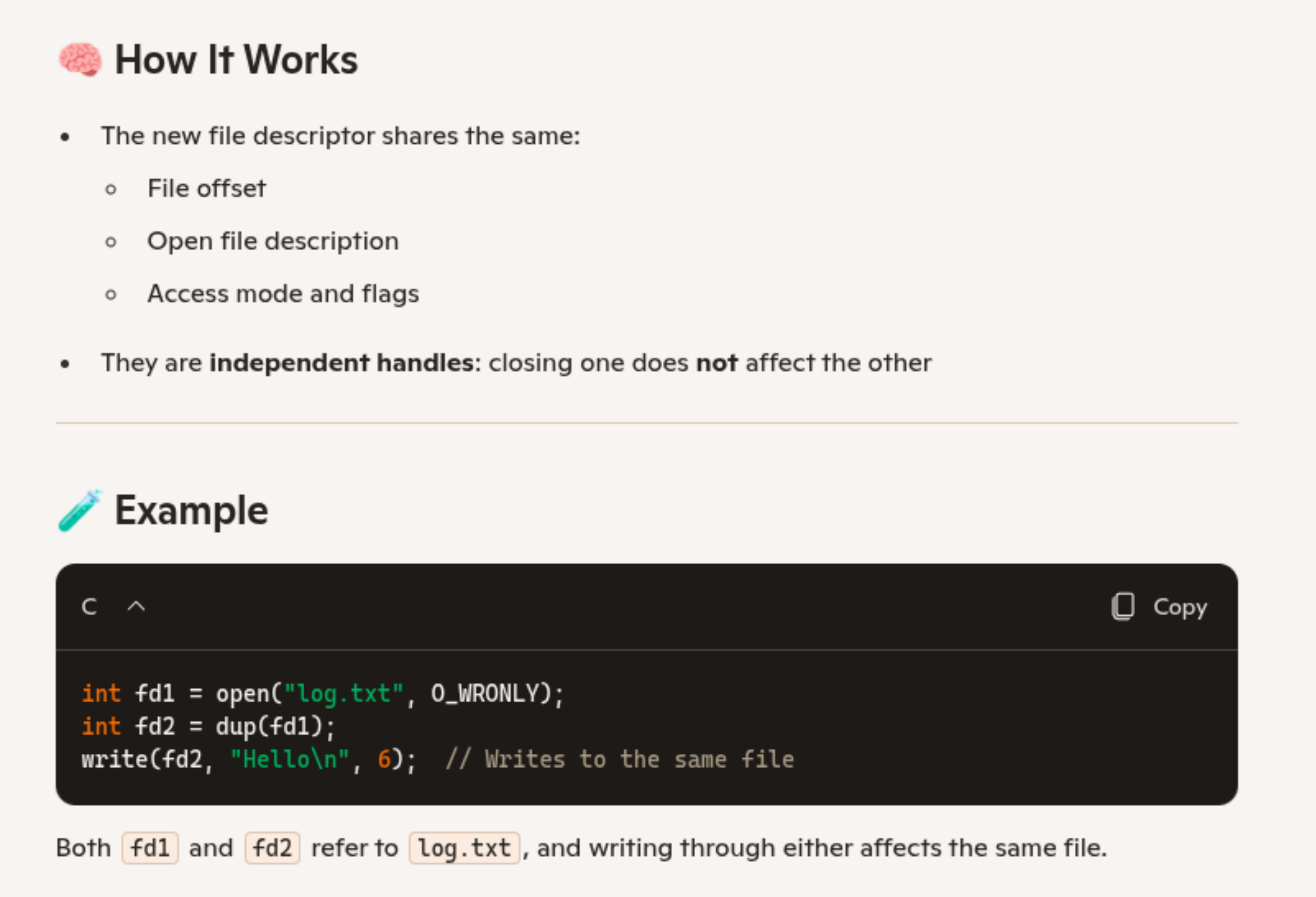
linux, dup system call

The `dup()` system call in Linux is used to **duplicate an existing file descriptor**, creating a new one that refers to the same open file, pipe, or socket.

📘 Function Prototype

```c
#include <unistd.h>

int dup(int oldfd);
```

- `oldfd`: the file descriptor to duplicate
- Returns: new file descriptor (lowest unused number), or `-1` on error



🧠 How It Works

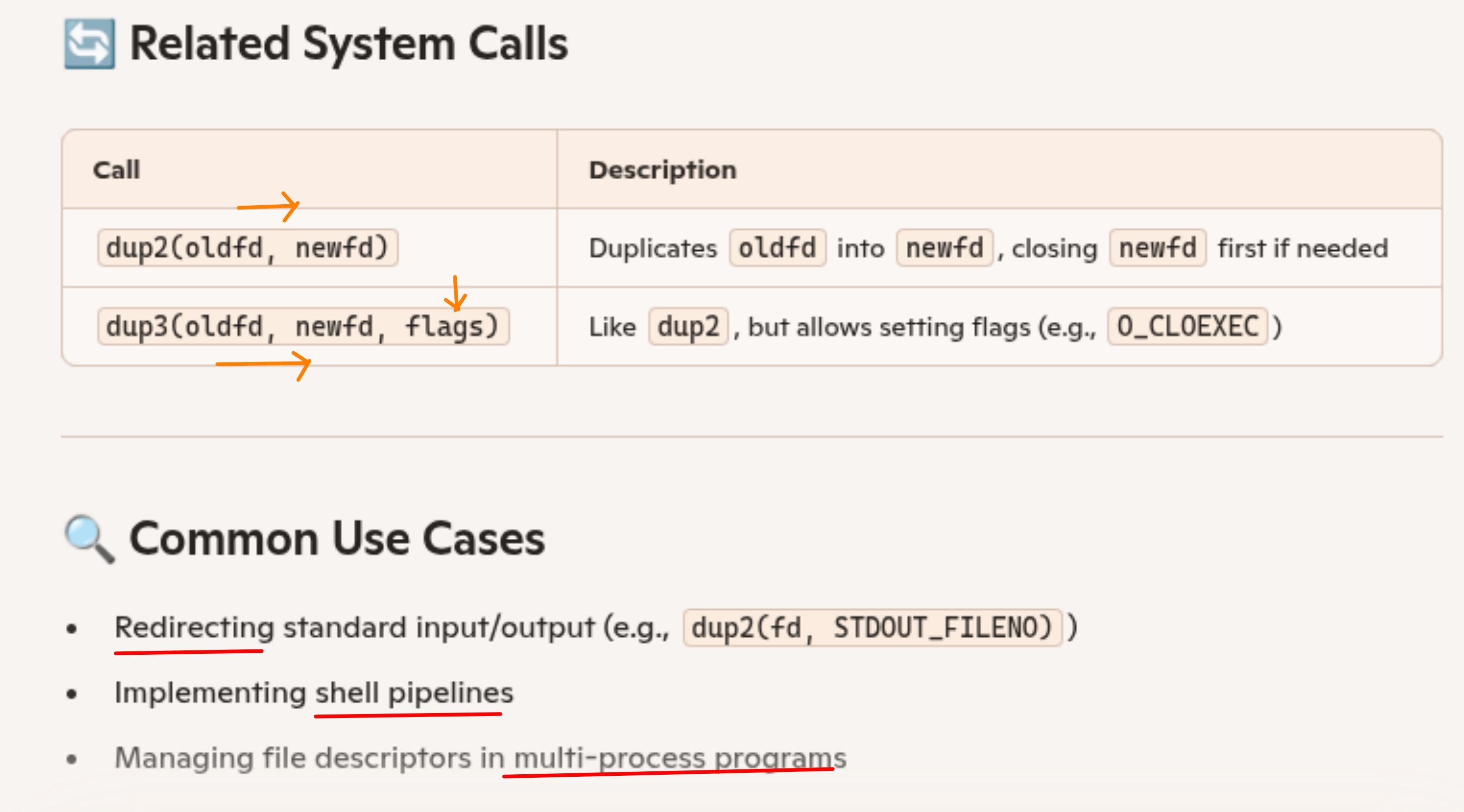
- The new file descriptor shares the same:
    - File offset
    - Open file description
    - Access mode and flags
- They are **independent handles**: closing one does **not** affect the other

---

🧪 Example

```c
int fd1 = open("log.txt", O_WRONLY);
int fd2 = dup(fd1);
write(fd2, "Hello\n", 6);  // Writes to the same file
```

Both `fd1` and `fd2` refer to `log.txt`, and writing through either affects the same file.

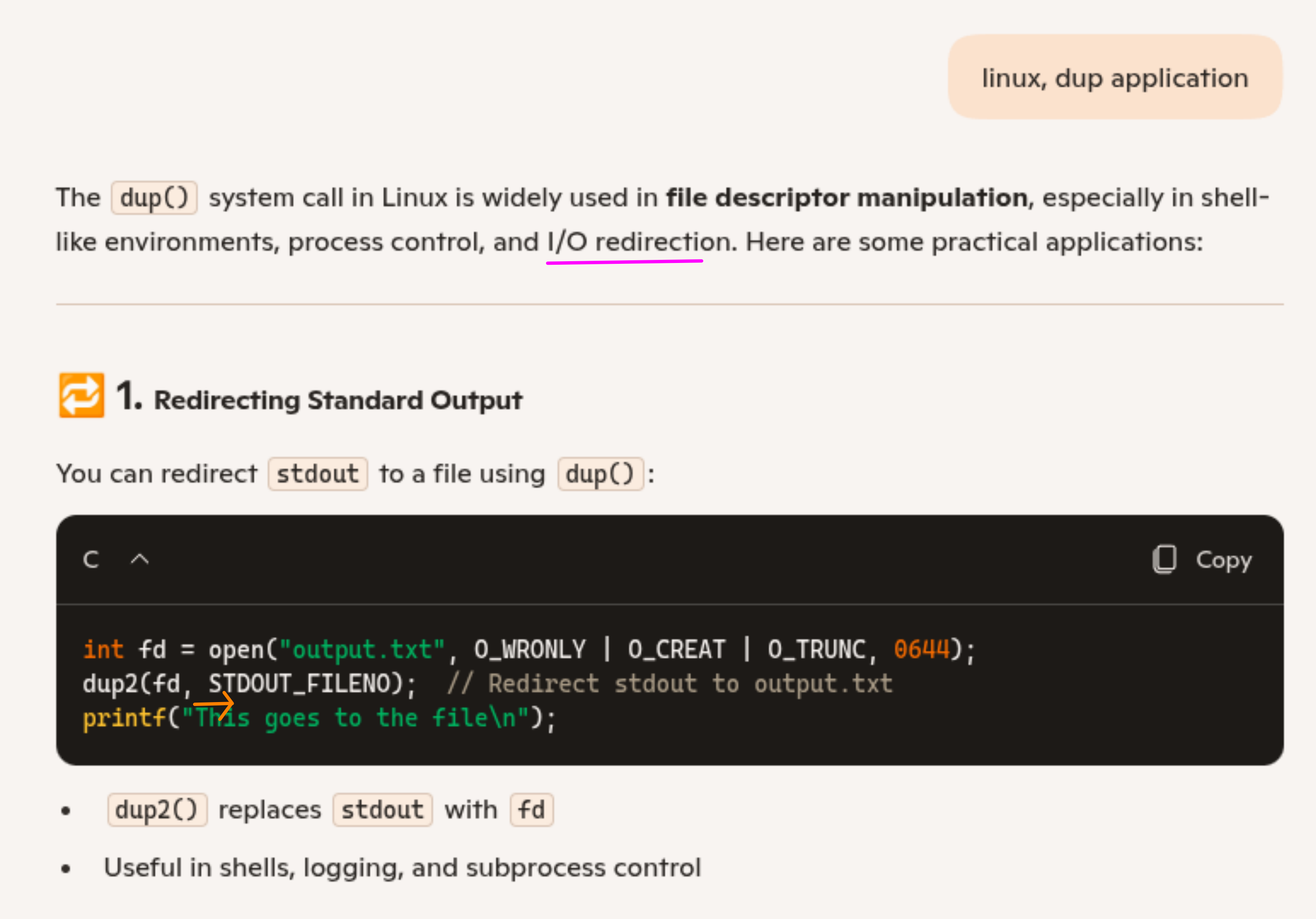


🔄 Related System Calls

Call	Description
`dup2(oldfd, newfd)`	Duplicates `oldfd` into `newfd`, closing `newfd` first if needed
`dup3(oldfd, newfd, flags)`	Like `dup2`, but allows setting flags (e.g., `O_CLOEXEC`)

🔍 Common Use Cases

- Redirecting standard input/output (e.g., `dup2(fd, STDOUT_FILENO)`)
- Implementing shell pipelines
- Managing file descriptors in multi-process programs



linux, dup application

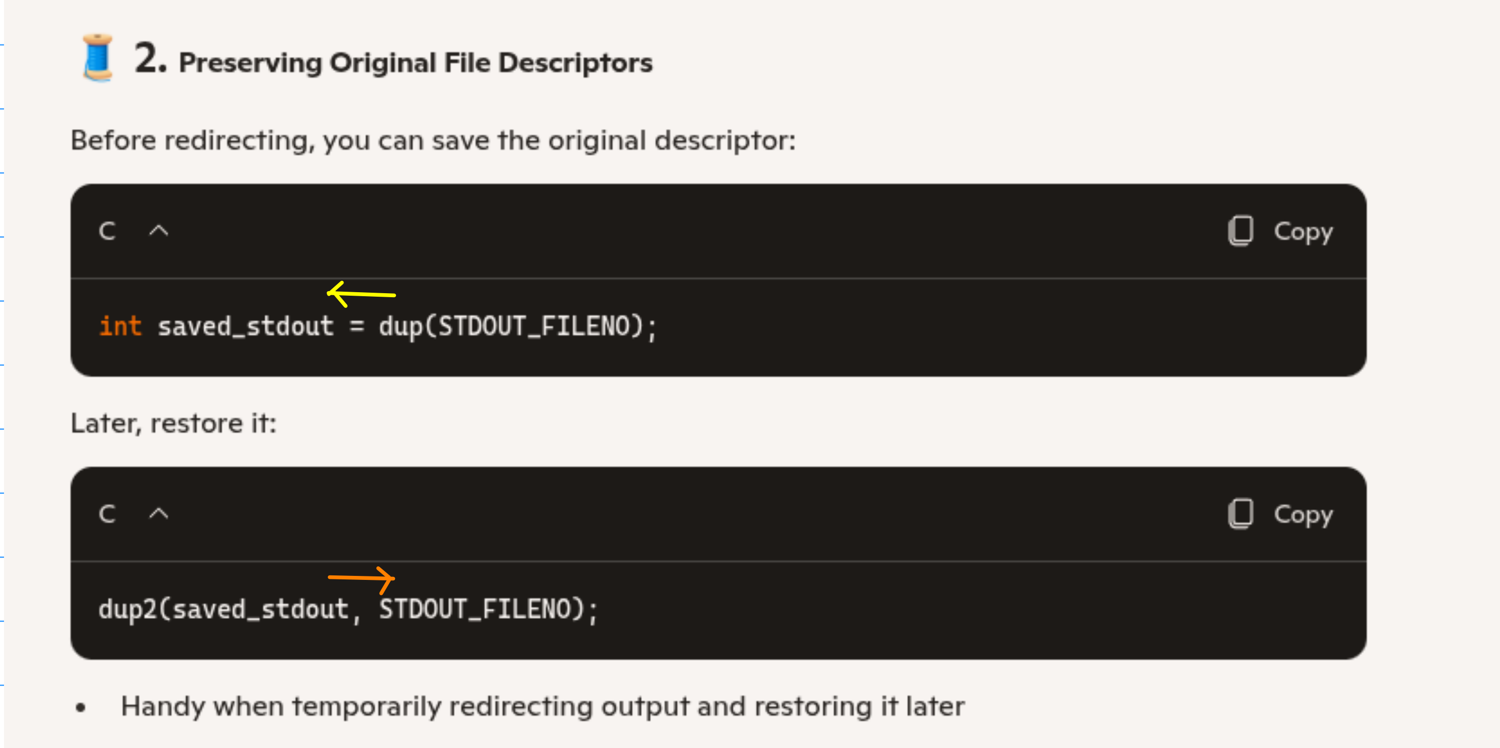
The `dup()` system call in Linux is widely used in **file descriptor manipulation**, especially in shell-like environments, process control, and I/O redirection. Here are some practical applications:

🔁 1. Redirecting Standard Output

You can redirect `stdout` to a file using `dup()`:

```c
int fd = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
dup2(fd, STDOUT_FILENO);  // Redirect stdout to output.txt
printf("This goes to the file\n");
```

- `dup2()` replaces `stdout` with `fd`
- Useful in shells, logging, and subprocess control



🧵 2. Preserving Original File Descriptors

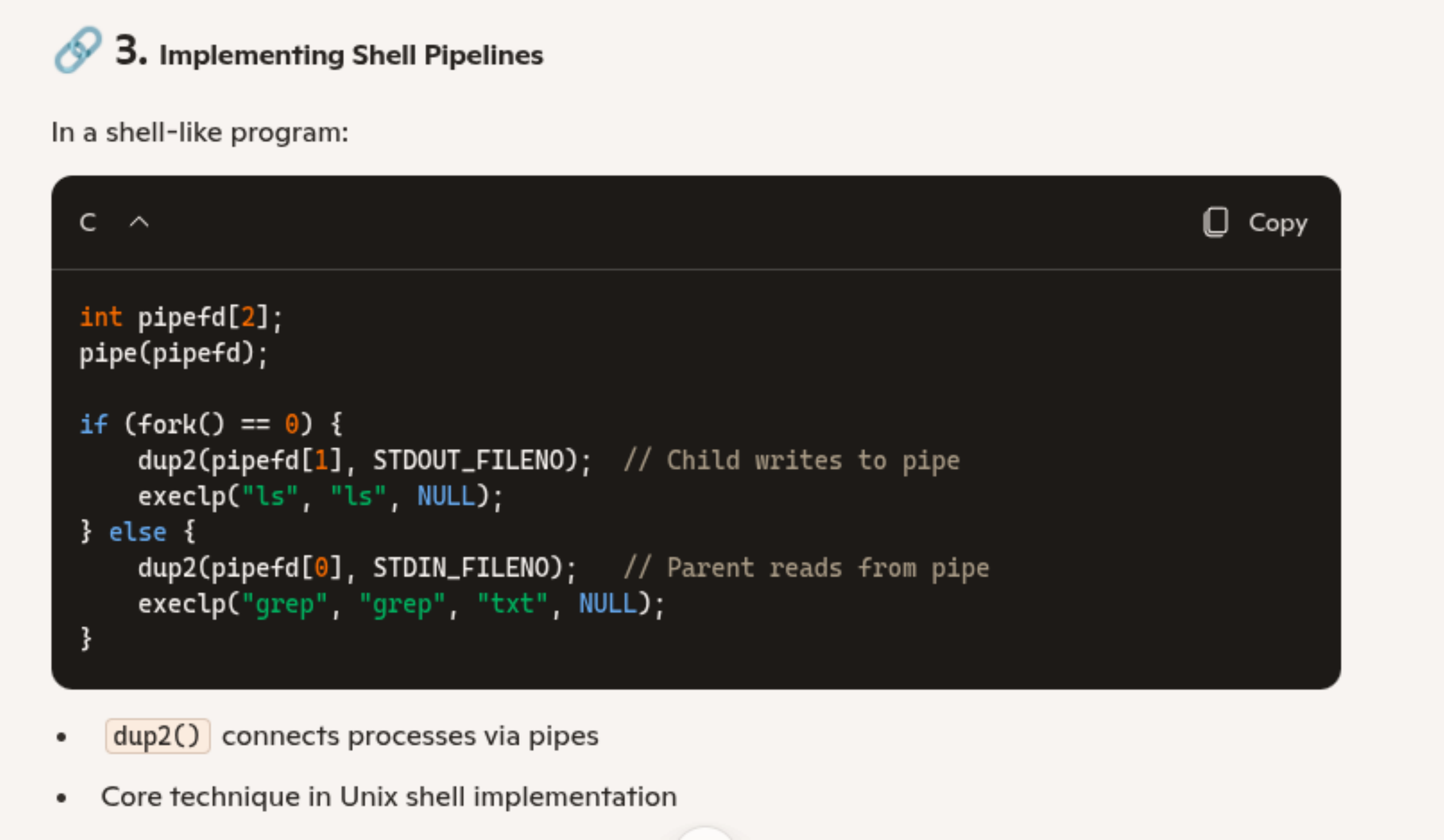
Before redirecting, you can save the original descriptor:

```c
int saved_stdout = dup(STDOUT_FILENO);
```

Later, restore it:

```c
dup2(saved_stdout, STDOUT_FILENO);
```

- Handy when temporarily redirecting output and restoring it later

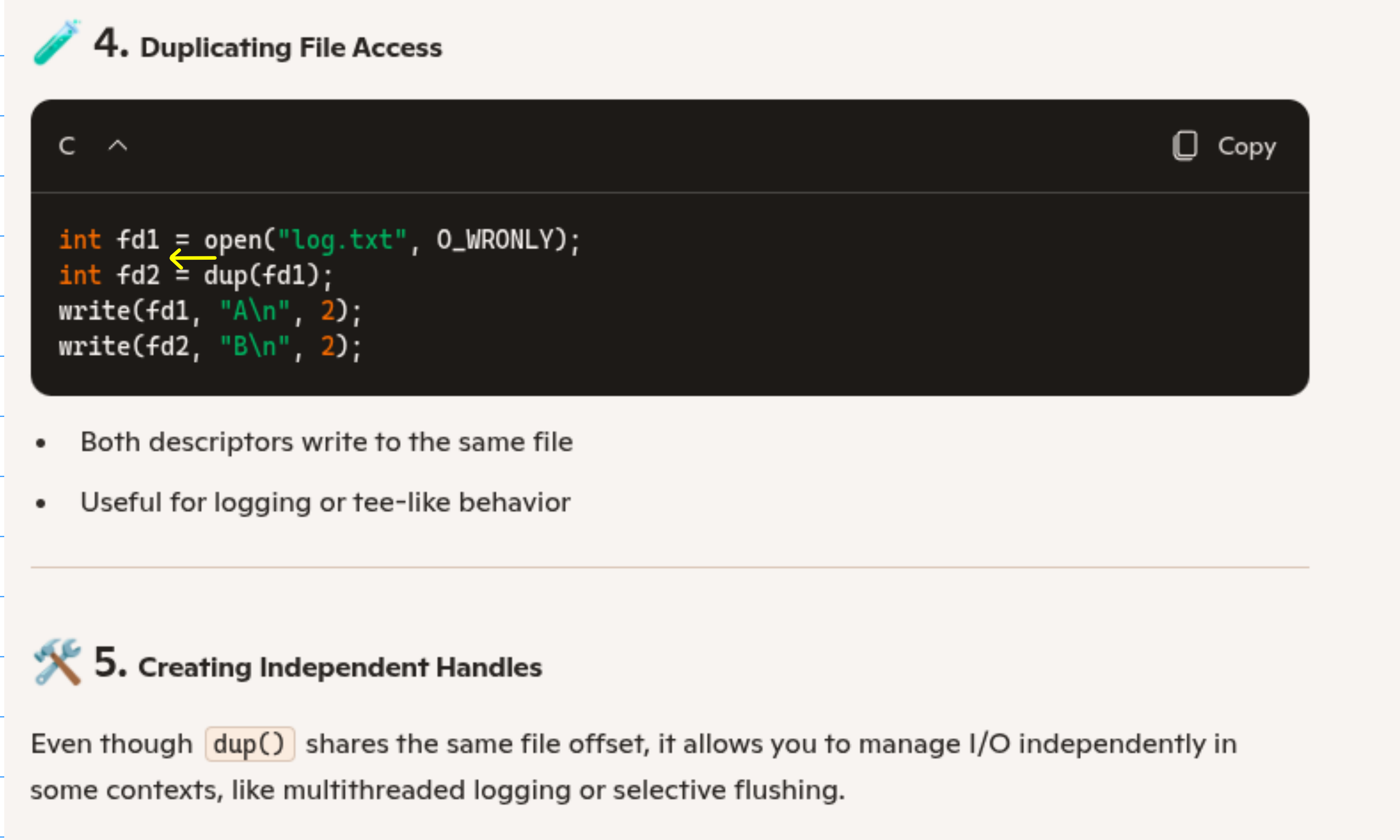


🔗 3. Implementing Shell Pipelines

In a shell-like program:

```c
int pipefd[2];
pipe(pipefd);

if (fork() == 0) {
    dup2(pipefd[1], STDOUT_FILENO);  // Child writes to pipe
    execlp("ls", "ls", NULL);
} else {
    dup2(pipefd[0], STDIN_FILENO);   // Parent reads from pipe
    execlp("grep", "grep", "txt", NULL);
}
```

- `dup2()` connects processes via pipes
- Core technique in Unix shell implementation



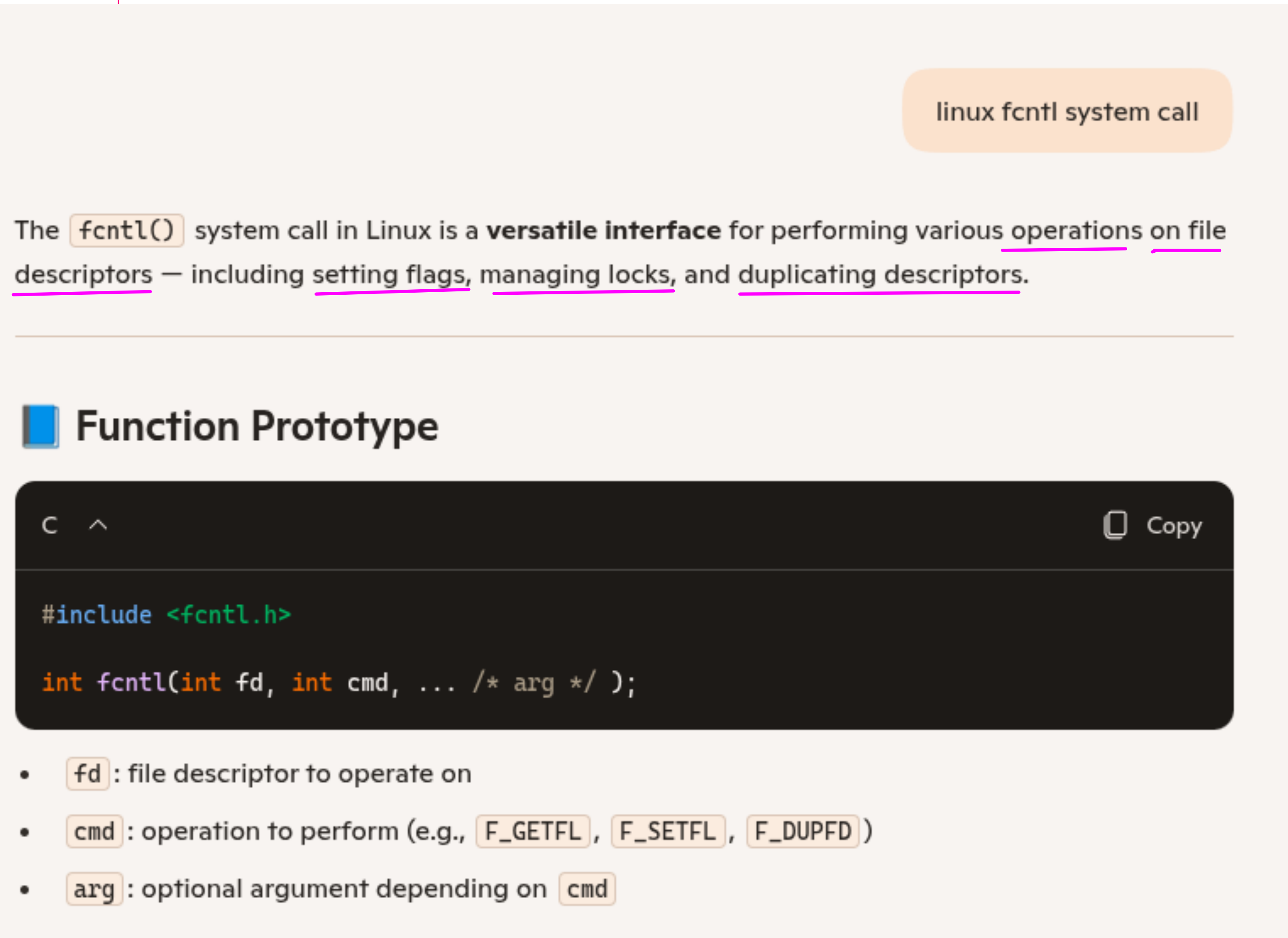
🧪 4. Duplicating File Access

```c
int fd1 = open("log.txt", O_WRONLY);
int fd2 = dup(fd1);
write(fd1, "A\n", 2);
write(fd2, "B\n", 2);
```

- Both descriptors write to the same file
- Useful for logging or tee-like behavior

🛠️ 5. Creating Independent Handles

Even though `dup()` shares the same file offset, it allows you to manage I/O independently in some contexts, like multithreaded logging or selective flushing.



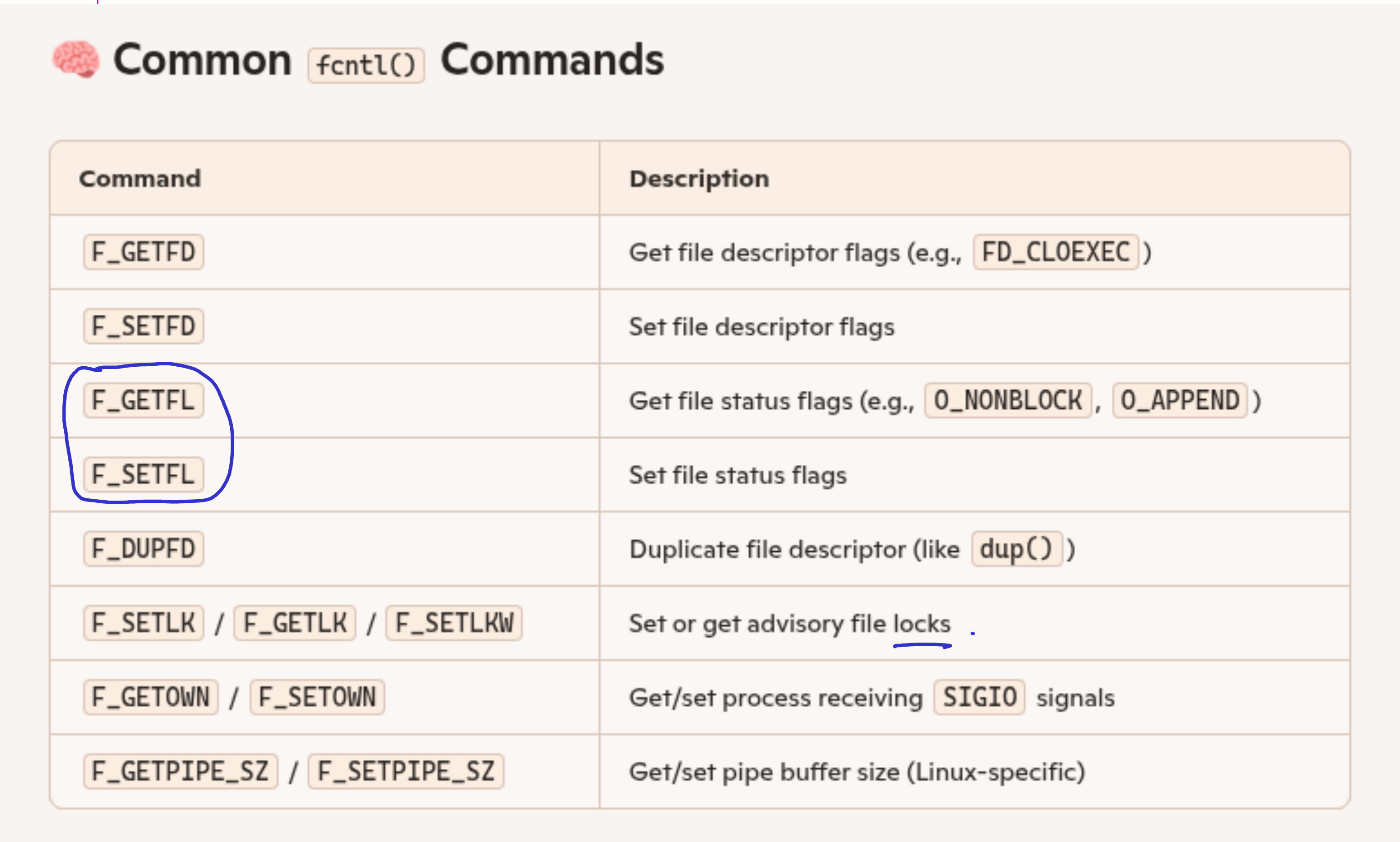
linux fcntl system call

The `fcntl()` system call in Linux is a **versatile interface** for performing various operations on file descriptors — including setting flags, managing locks, and duplicating descriptors.

📘 Function Prototype

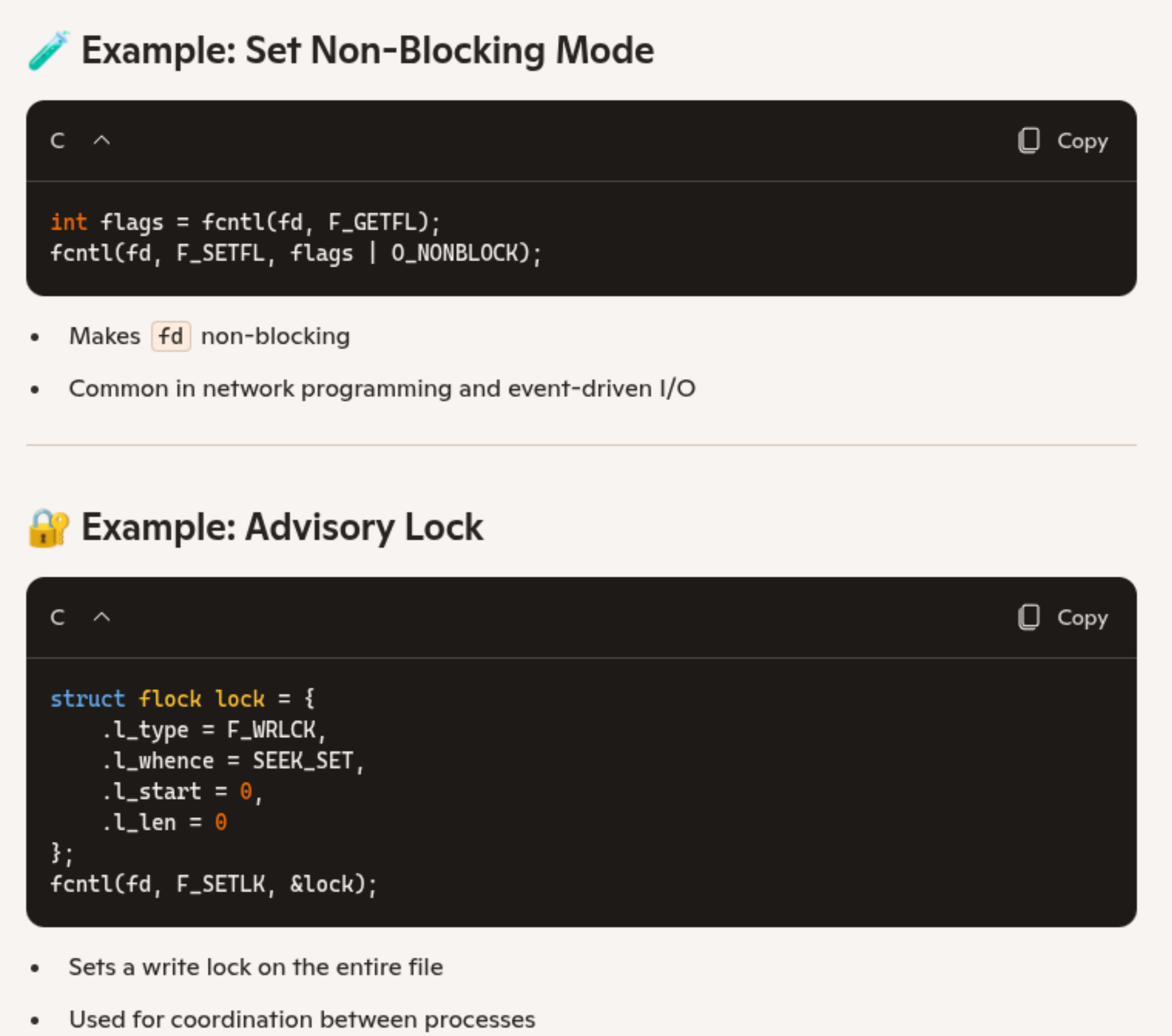
```c
#include <fcntl.h>

int fcntl(int fd, int cmd, ... /* arg */ );
```

- `fd` : file descriptor to operate on
- `cmd` : operation to perform (e.g., `F_GETFL`, `F_SETFL`, `F_DUPFD`)
- `arg` : optional argument depending on `cmd`



🧠 Common `fcntl()` Commands

Command	Description
`F_GETFD`	Get file descriptor flags (e.g., `FD_CLOEXEC`)
`F_SETFD`	Set file descriptor flags
`F_GETFL`	Get file status flags (e.g., `O_NONBLOCK`, `O_APPEND`)
`F_SETFL`	Set file status flags
`F_DUPFD`	Duplicate file descriptor (like `dup()`)
`F_SETLK` / `F_GETLK` / `F_SETLKW`	Set or get advisory file locks
`F_GETOWN` / `F_SETOWN`	Get/set process receiving `SIGIO` signals
`F_GETPIPE_SZ` / `F_SETPIPE_SZ`	Get/set pipe buffer size (Linux-specific)



🧪 Example: Set Non-Blocking Mode

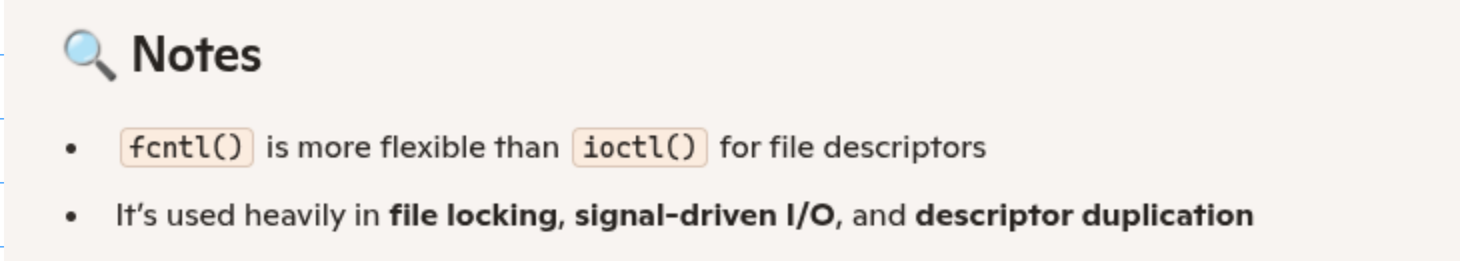
```c
int flags = fcntl(fd, F_GETFL);
fcntl(fd, F_SETFL, flags | O_NONBLOCK);
```

- Makes `fd` non-blocking
- Common in network programming and event-driven I/O

🔐 Example: Advisory Lock

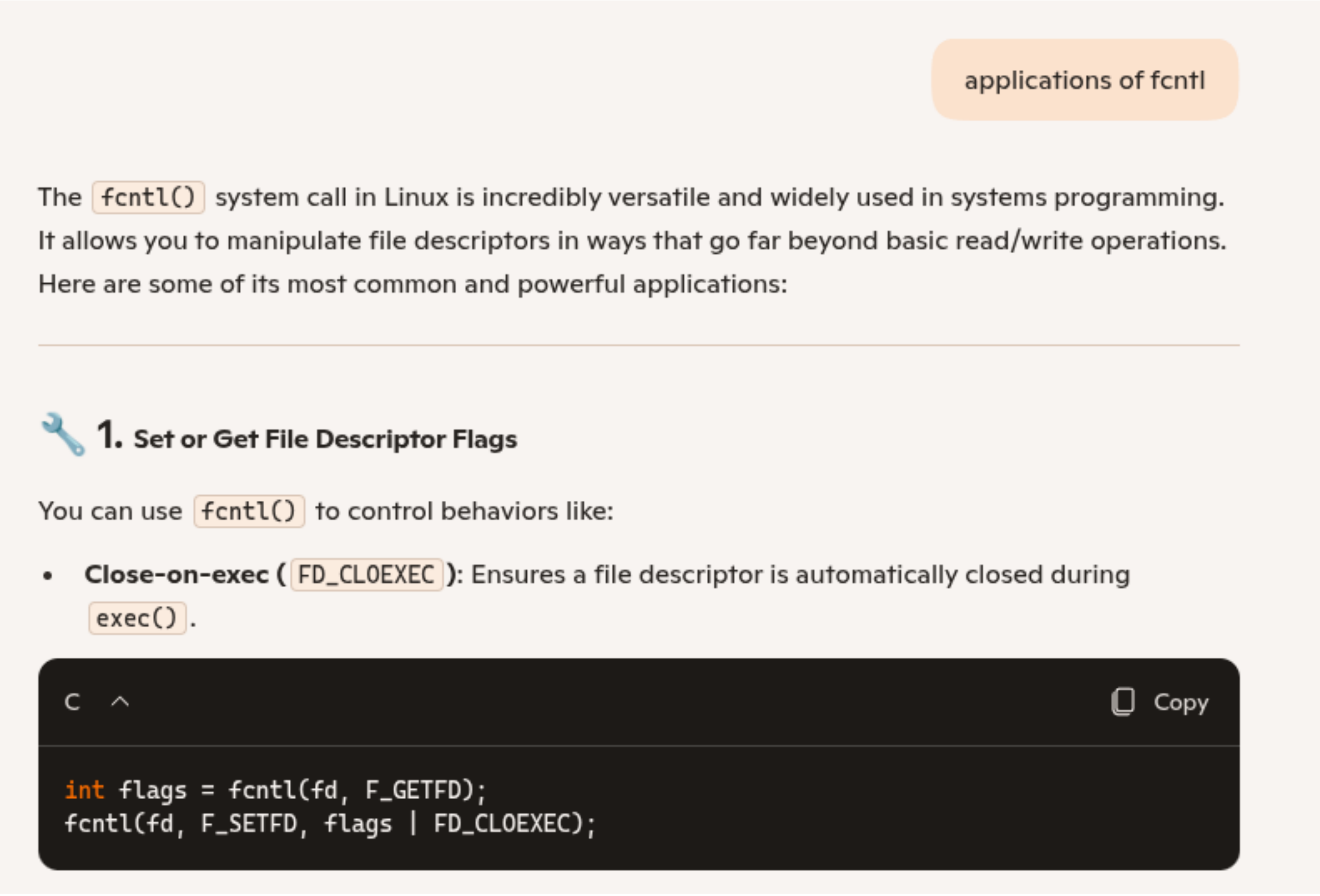
```c
struct flock lock = {
    .l_type = F_WRLCK,
    .l_whence = SEEK_SET,
    .l_start = 0,
    .l_len = 0
};
fcntl(fd, F_SETLK, &lock);
```

- Sets a write lock on the entire file
- Used for coordination between processes



🔍 Notes

- `fcntl()` is more flexible than `ioctl()` for file descriptors
- It's used heavily in **file locking, signal-driven I/O**, and **descriptor duplication**



applications of fcntl

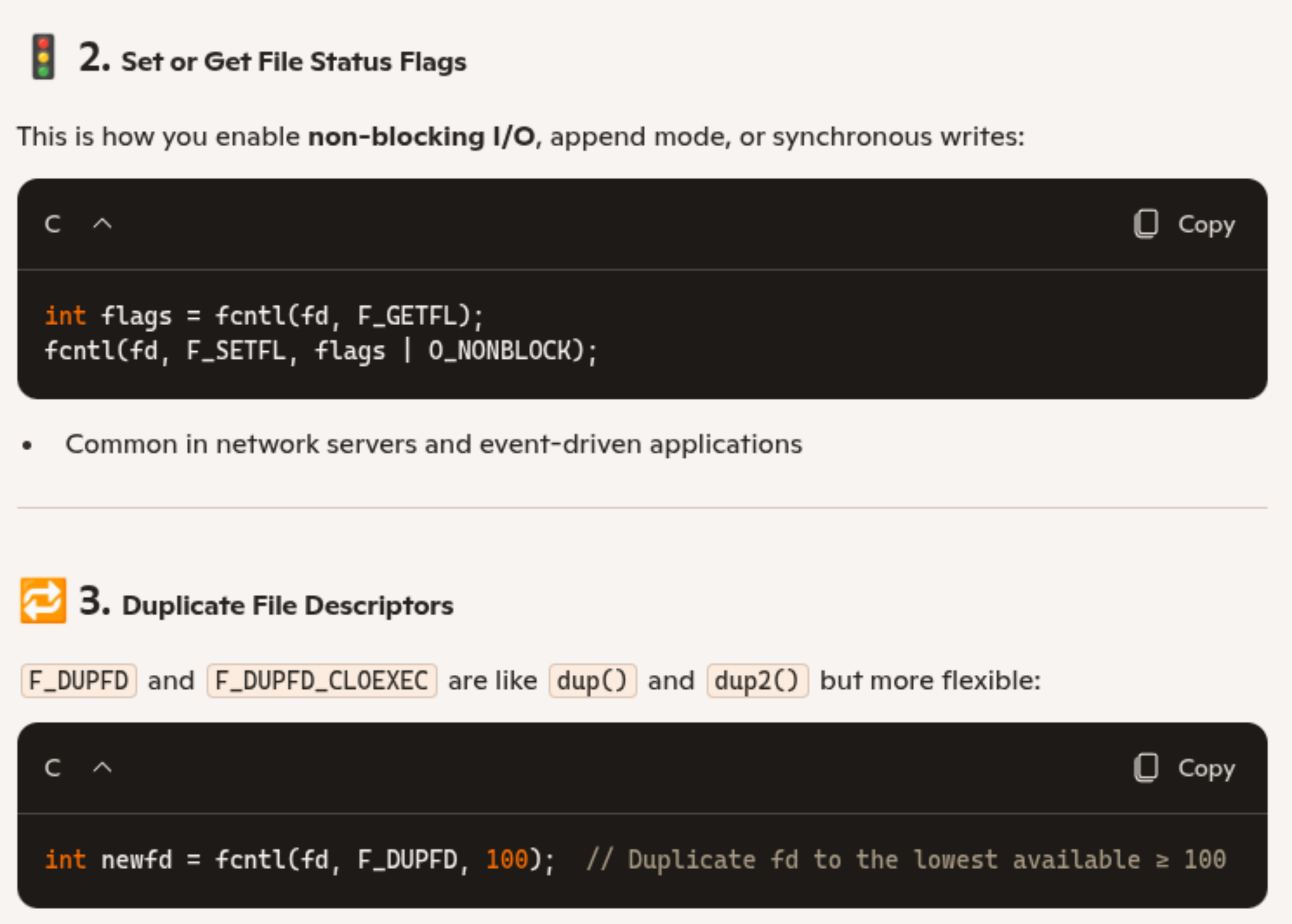
The `fcntl()` system call in Linux is incredibly versatile and widely used in systems programming. It allows you to manipulate file descriptors in ways that go far beyond basic read/write operations. Here are some of its most common and powerful applications:

🔧 1. Set or Get File Descriptor Flags

You can use `fcntl()` to control behaviors like:

- **Close-on-exec (** `FD_CLOEXEC` **)**: Ensures a file descriptor is automatically closed during `exec()`.

```c
int flags = fcntl(fd, F_GETFD);
fcntl(fd, F_SETFD, flags | FD_CLOEXEC);
```



🚦 2. Set or Get File Status Flags

This is how you enable **non-blocking I/O**, append mode, or synchronous writes:

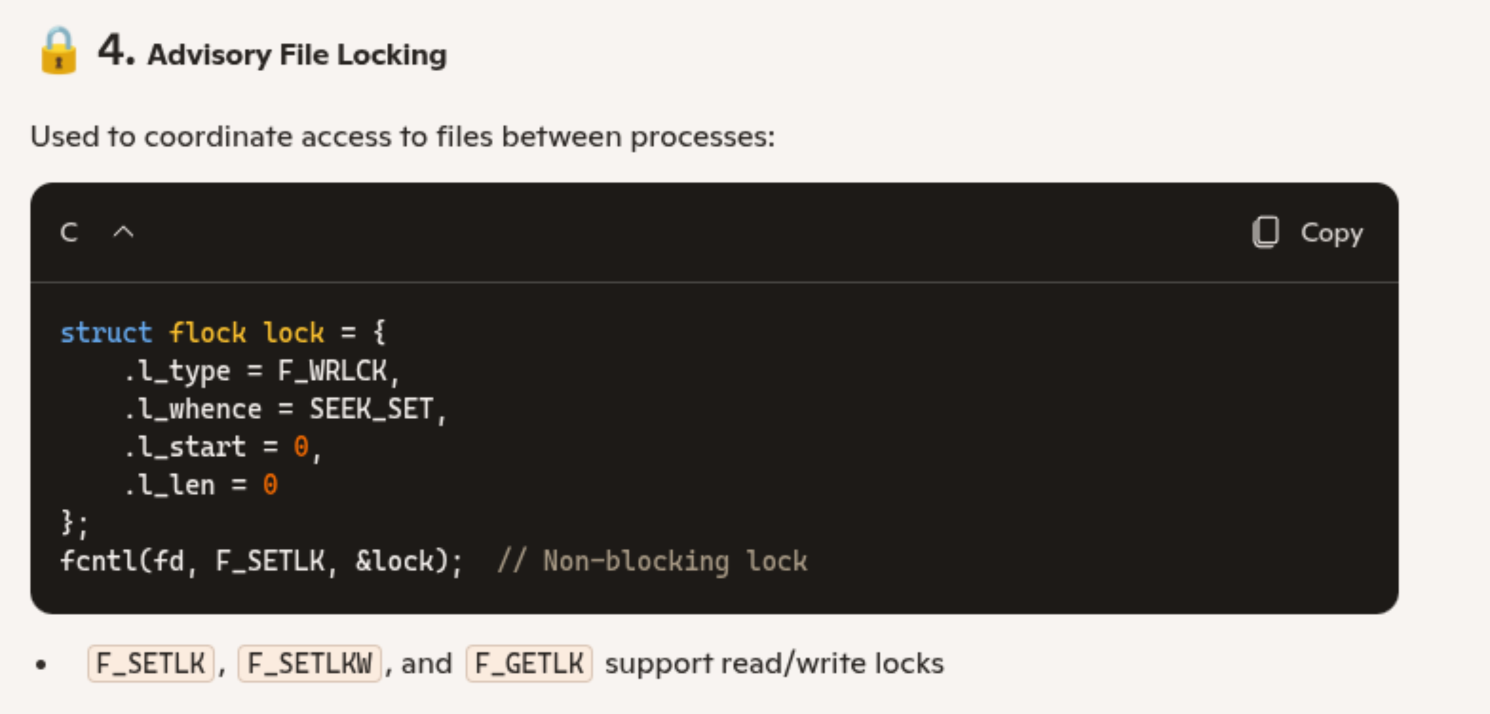
```c
int flags = fcntl(fd, F_GETFL);
fcntl(fd, F_SETFL, flags | O_NONBLOCK);
```

- Common in network servers and event-driven applications

🔁 3. Duplicate File Descriptors

`F_DUPFD` and `F_DUPFD_CLOEXEC` are like `dup()` and `dup2()` but more flexible:

```c
int newfd = fcntl(fd, F_DUPFD, 100);  // Duplicate fd to the lowest available ≥ 100
```

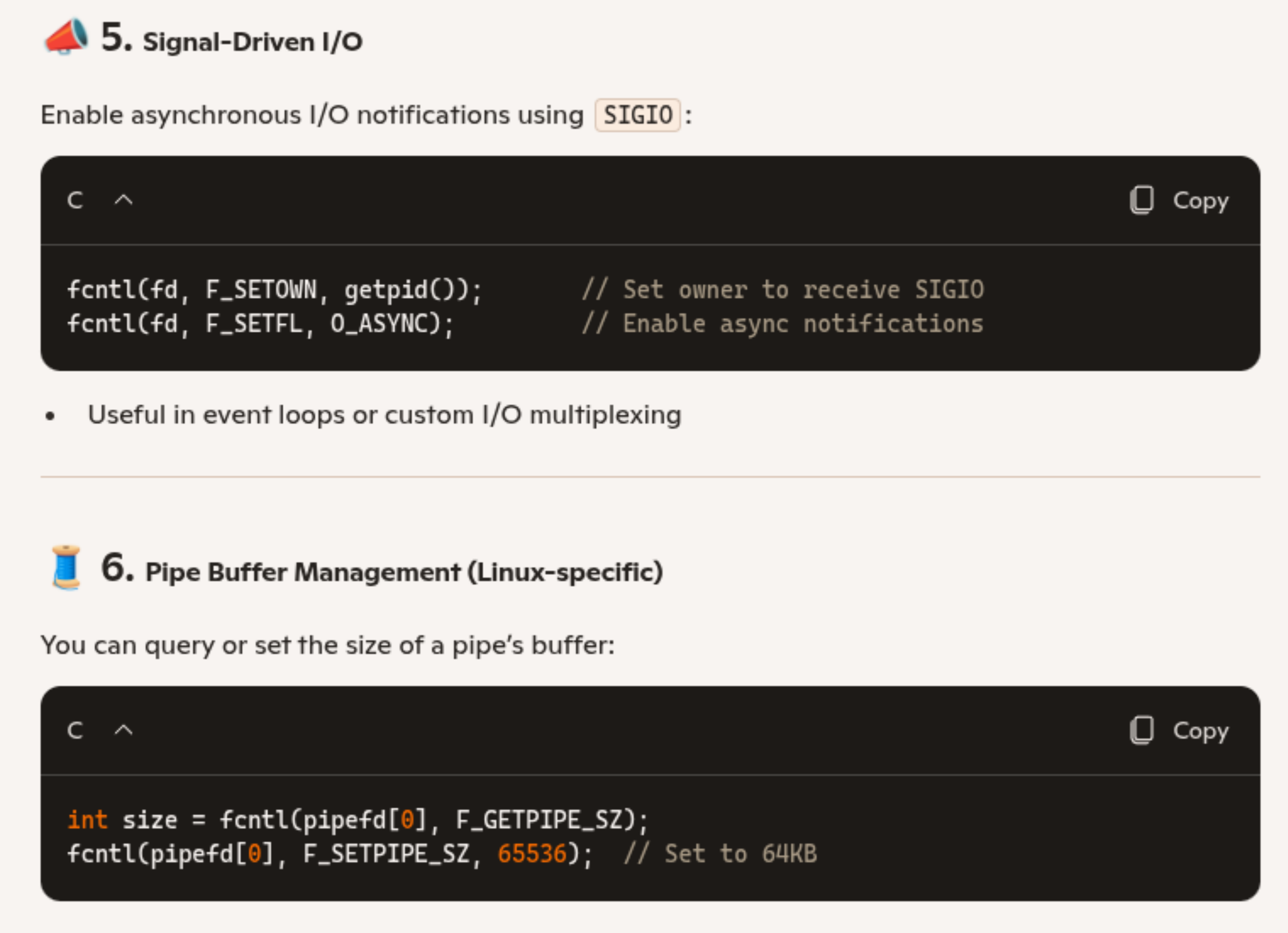


🔒 4. Advisory File Locking

Used to coordinate access to files between processes:

```c
struct flock lock = {
    .l_type = F_WRLCK,
    .l_whence = SEEK_SET,
    .l_start = 0,
    .l_len = 0
};
fcntl(fd, F_SETLK, &lock);  // Non-blocking lock
```

- `F_SETLK`, `F_SETLKW`, and `F_GETLK` support read/write locks



📣 5. Signal-Driven I/O

Enable asynchronous I/O notifications using `SIGIO`:

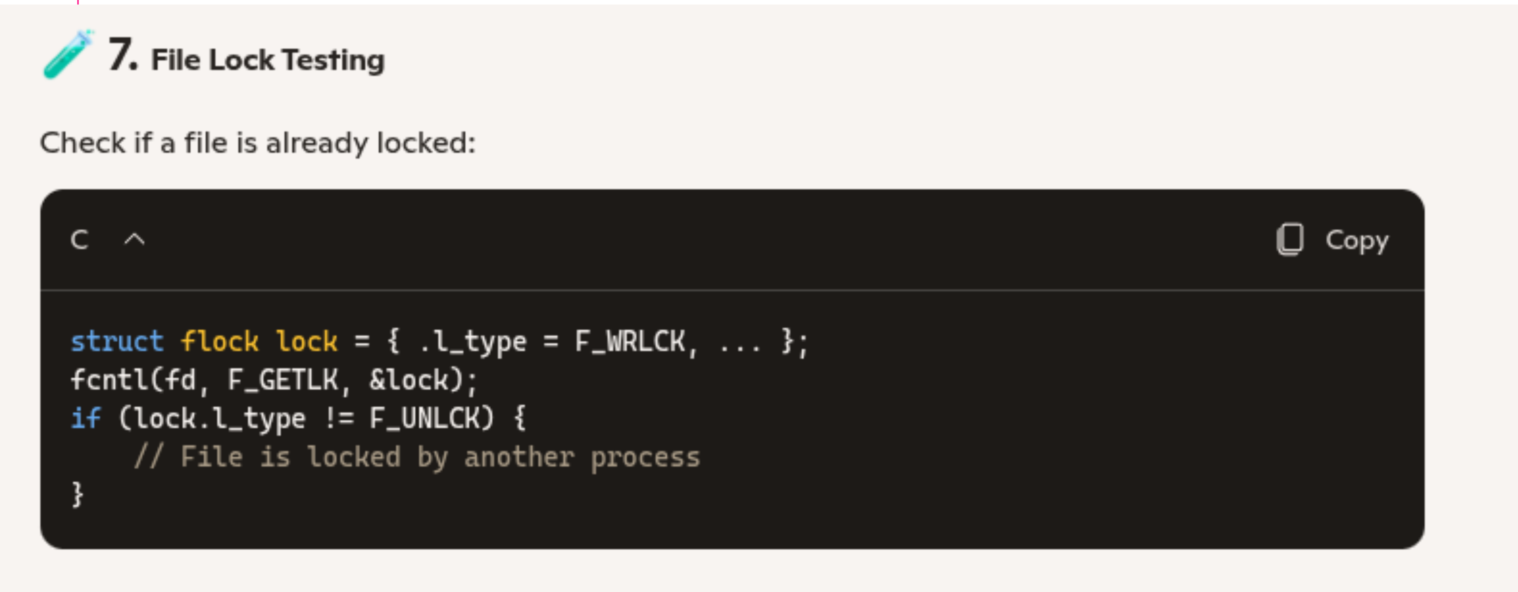
```c
fcntl(fd, F_SETOWN, getpid());        // Set owner to receive SIGIO
fcntl(fd, F_SETFL, O_ASYNC);          // Enable async notifications
```

- Useful in event loops or custom I/O multiplexing

🧵 6. Pipe Buffer Management (Linux-specific)

You can query or set the size of a pipe's buffer:

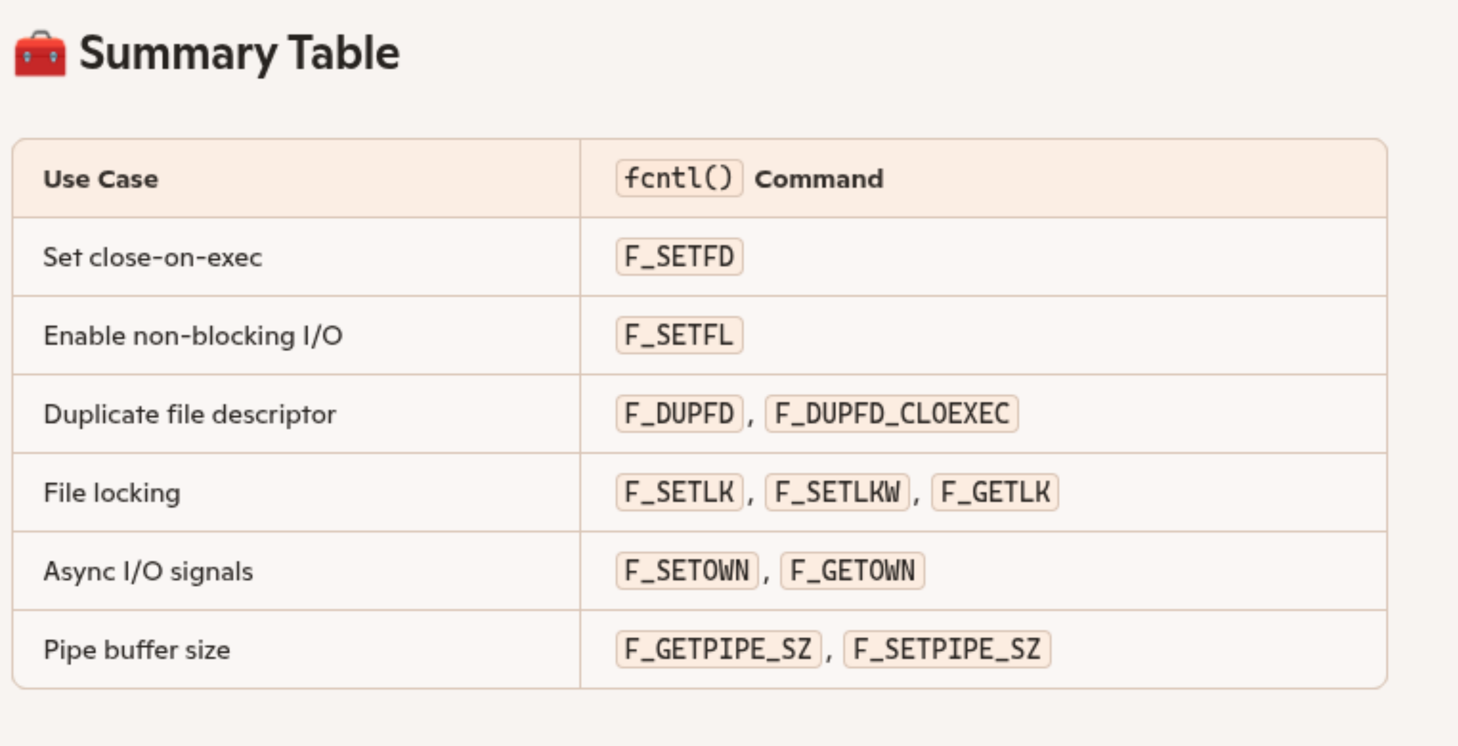
```c
int size = fcntl(pipefd[0], F_GETPIPE_SZ);
fcntl(pipefd[0], F_SETPIPE_SZ, 65536);  // Set to 64KB
```



🧪 7. File Lock Testing

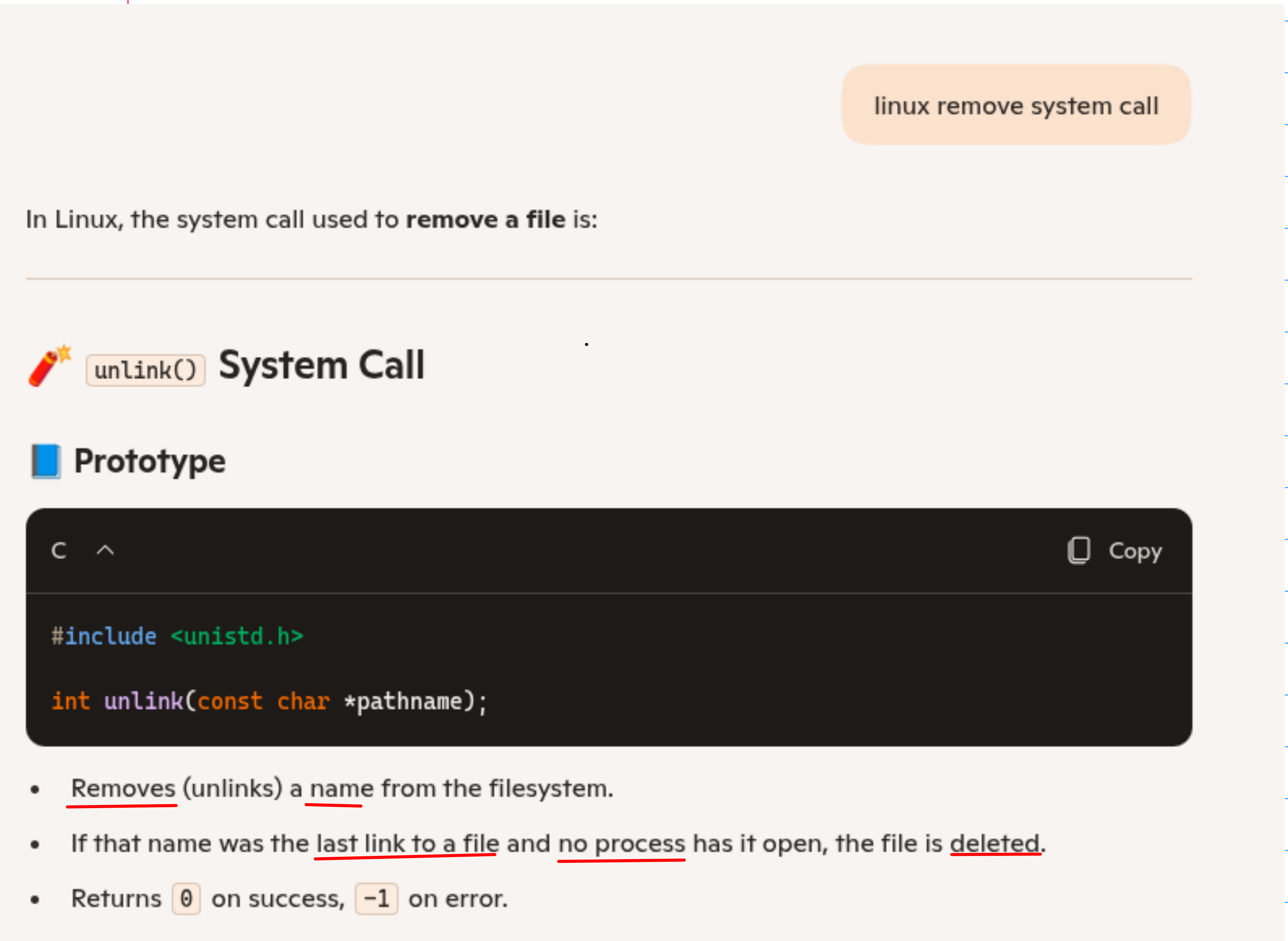
Check if a file is already locked:

```c
struct flock lock = { .l_type = F_WRLCK, ... };
fcntl(fd, F_GETLK, &lock);
if (lock.l_type != F_UNLCK) {
    // File is locked by another process
}
```



🧰 Summary Table

Use Case	`fcntl()` Command
Set close-on-exec	`F_SETFD`
Enable non-blocking I/O	`F_SETFL`
Duplicate file descriptor	`F_DUPFD`, `F_DUPFD_CLOEXEC`
File locking	`F_SETLK`, `F_SETLKW`, `F_GETLK`
Async I/O signals	`F_SETOWN`, `F_GETOWN`
Pipe buffer size	`F_GETPIPE_SZ`, `F_SETPIPE_SZ`



linux remove system call

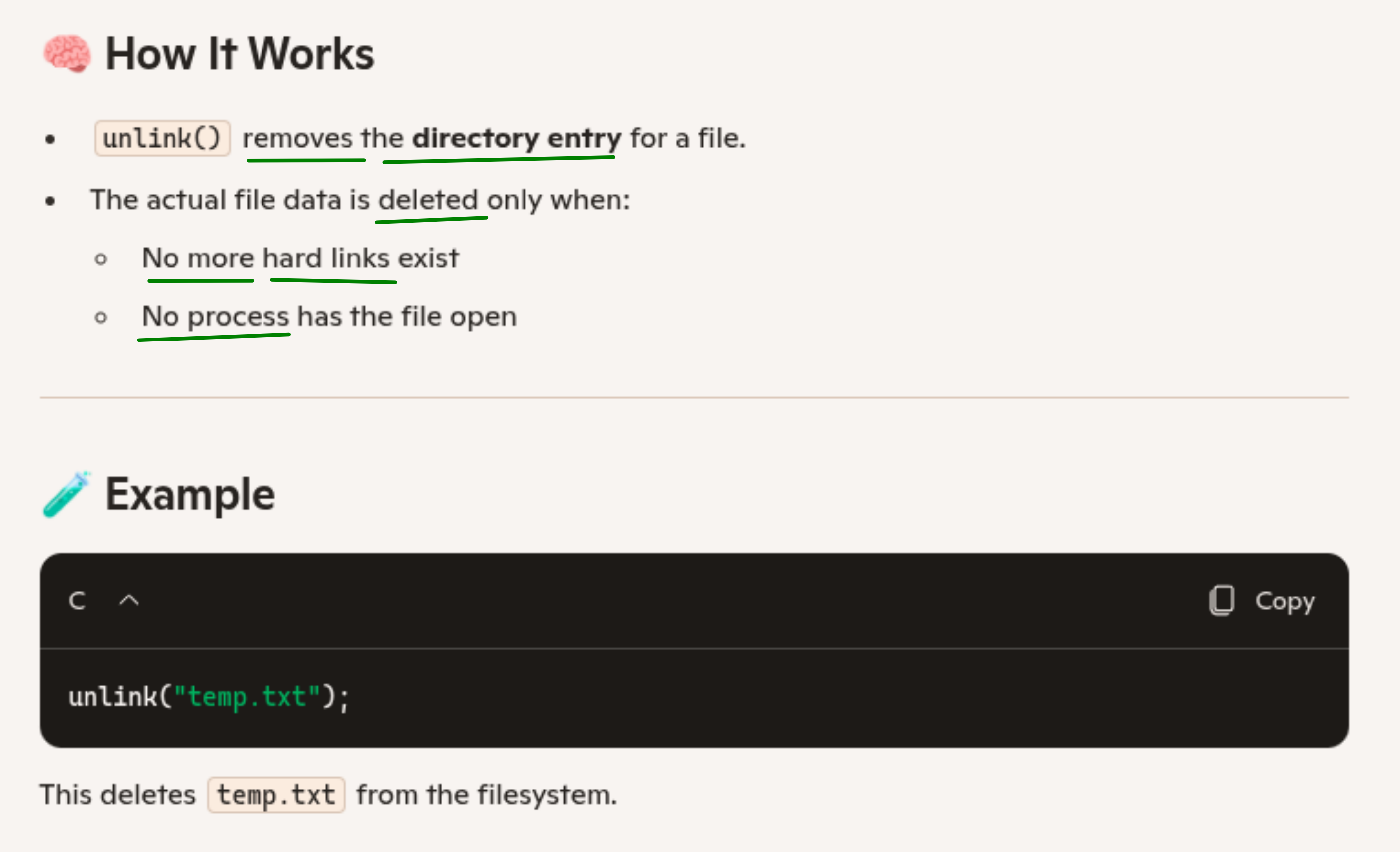
In Linux, the system call used to **remove a file** is:

🧨 `unlink()` System Call

📘 Prototype

```c
#include <unistd.h>

int unlink(const char *pathname);
```

- Removes (unlinks) a name from the filesystem.
- If that name was the last link to a file and no process has it open, the file is deleted.
- Returns `0` on success, `-1` on error.



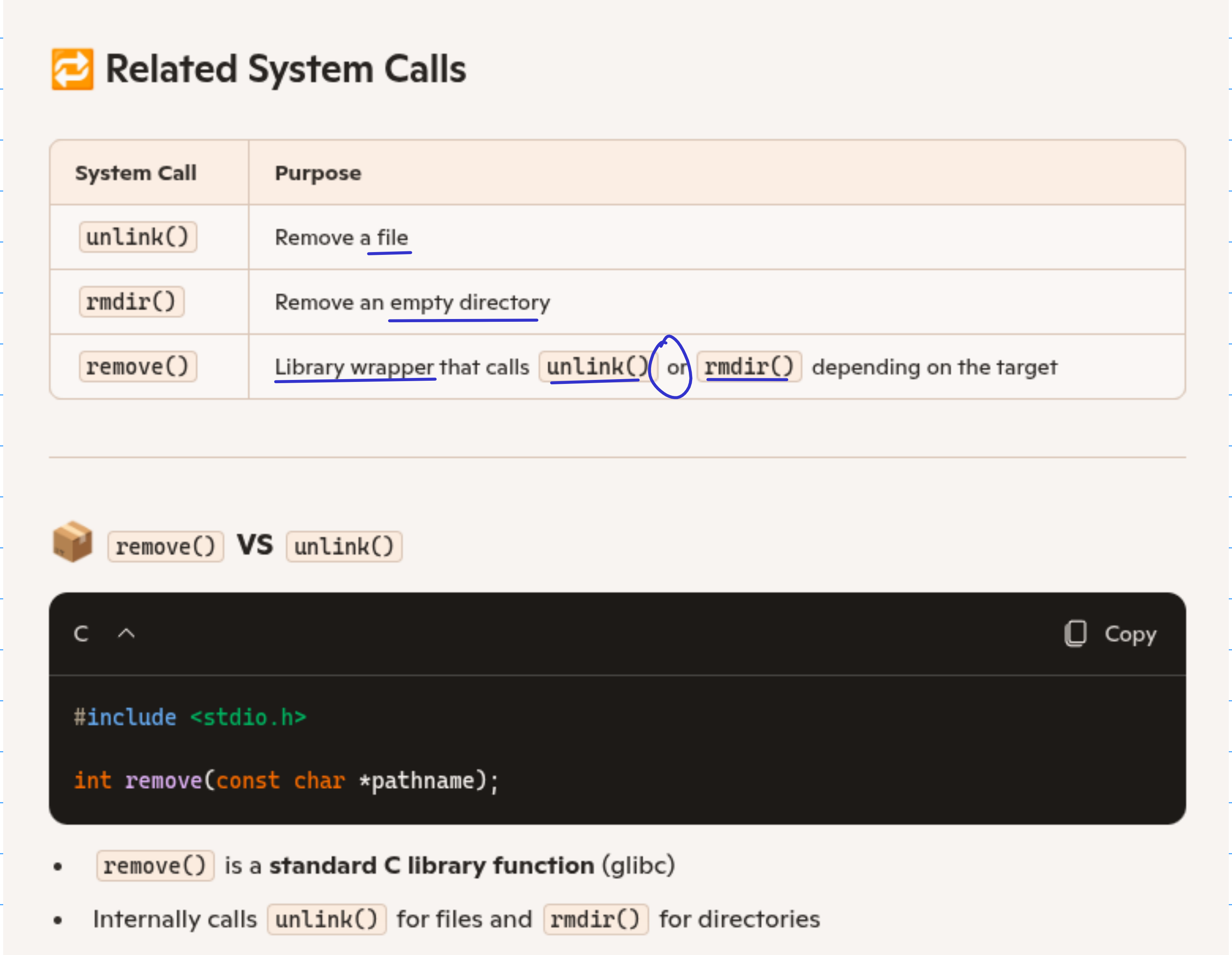
🧠 How It Works

- `unlink()` removes the **directory entry** for a file.
- The actual file data is deleted only when:
    - No more hard links exist
    - No process has the file open

🧪 Example

```c
unlink("temp.txt");
```

This deletes `temp.txt` from the filesystem.



🔁 Related System Calls

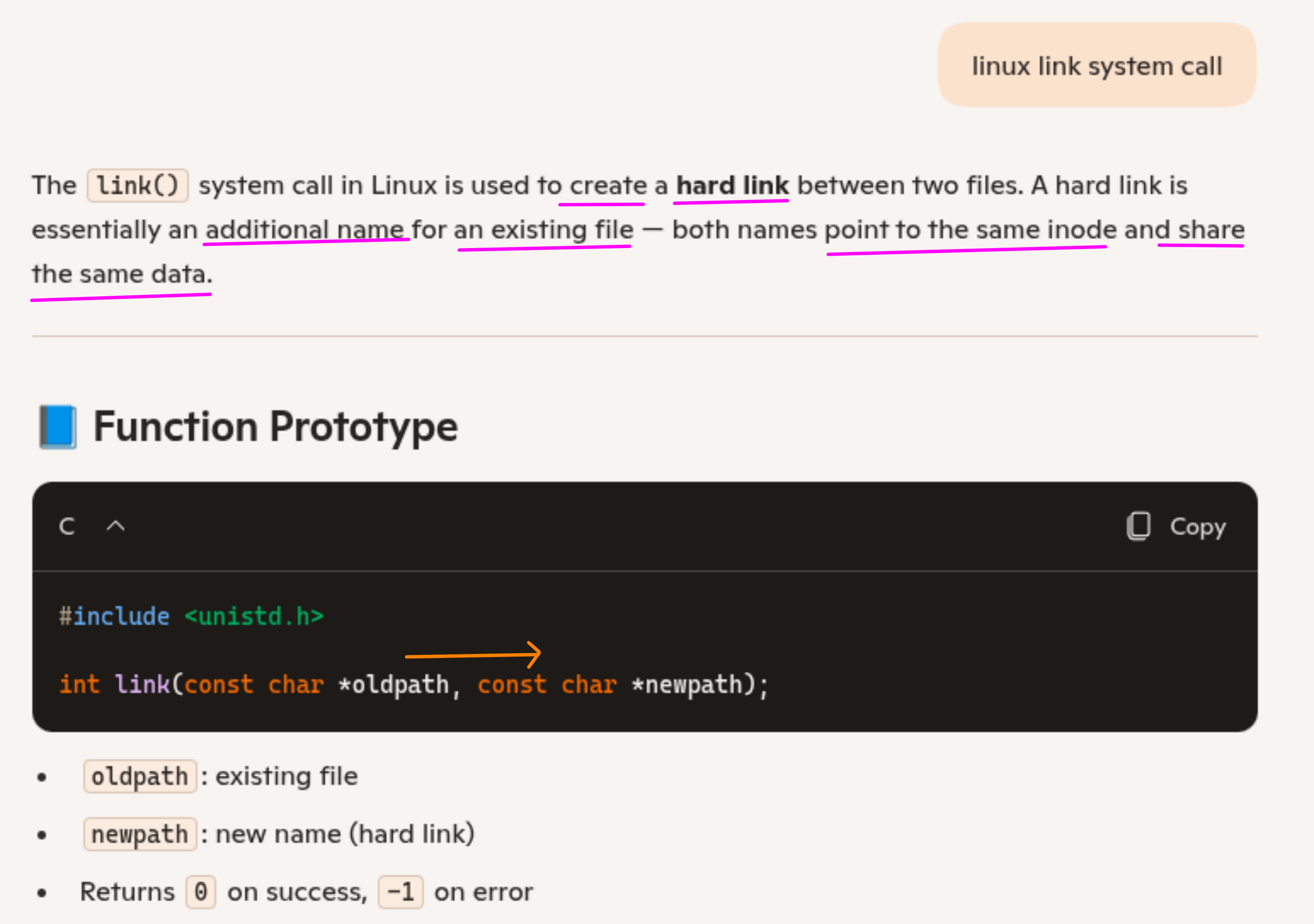
System Call	Purpose
`unlink()`	Remove a file
`rmdir()`	Remove an empty directory
`remove()`	Library wrapper that calls `unlink()` or `rmdir()` depending on the target

---

📦 `remove()` VS `unlink()`

```c
#include <stdio.h>

int remove(const char *pathname);
```

- `remove()` is a **standard C library function** (glibc)
- Internally calls `unlink()` for files and `rmdir()` for directories



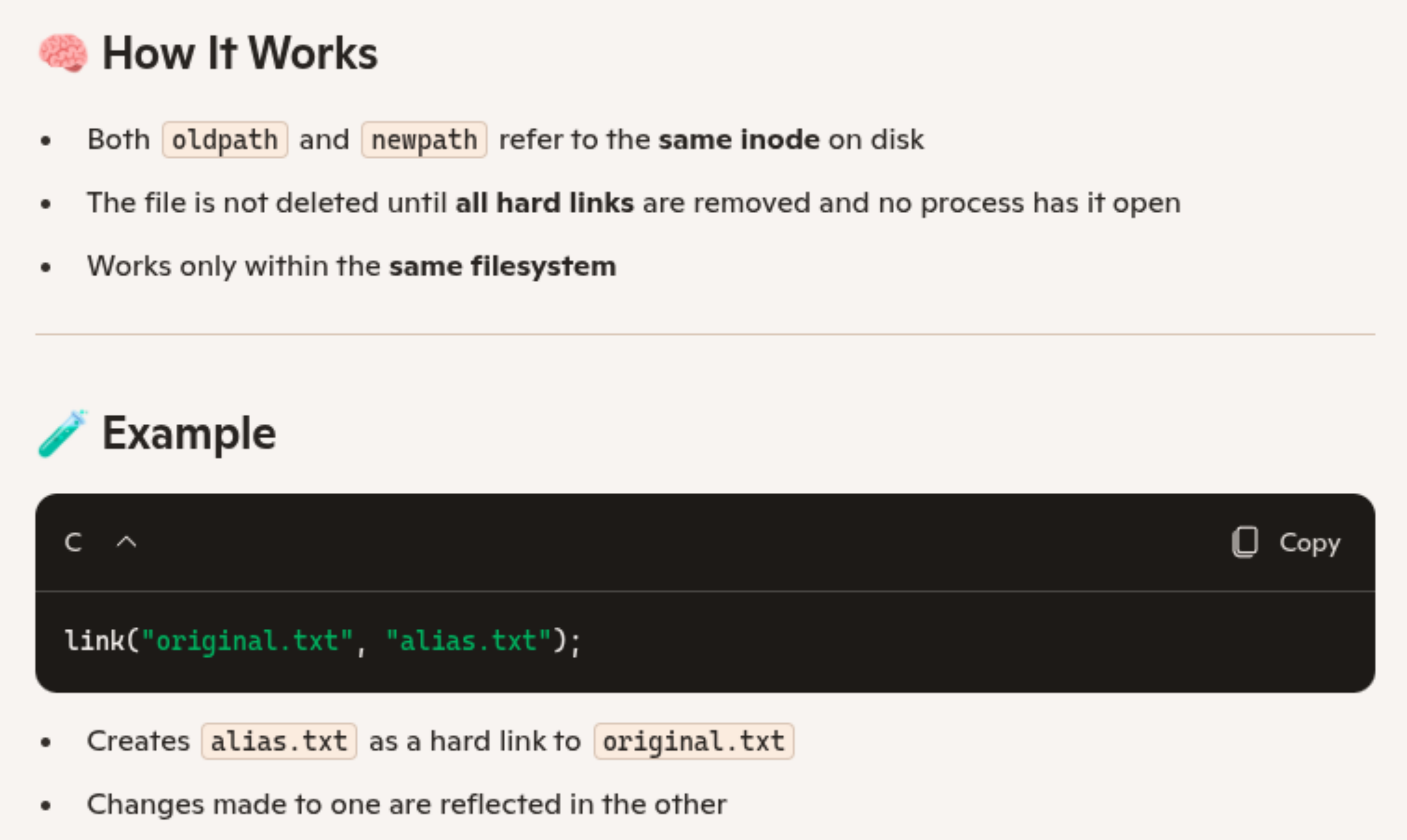
linux link system call

The `link()` system call in Linux is used to create a **hard link** between two files. A hard link is essentially an additional name for an existing file — both names point to the same inode and share the same data.

📘 Function Prototype

```c
#include <unistd.h>

int link(const char *oldpath, const char *newpath);
```

- `oldpath`: existing file
- `newpath`: new name (hard link)
- Returns `0` on success, `-1` on error



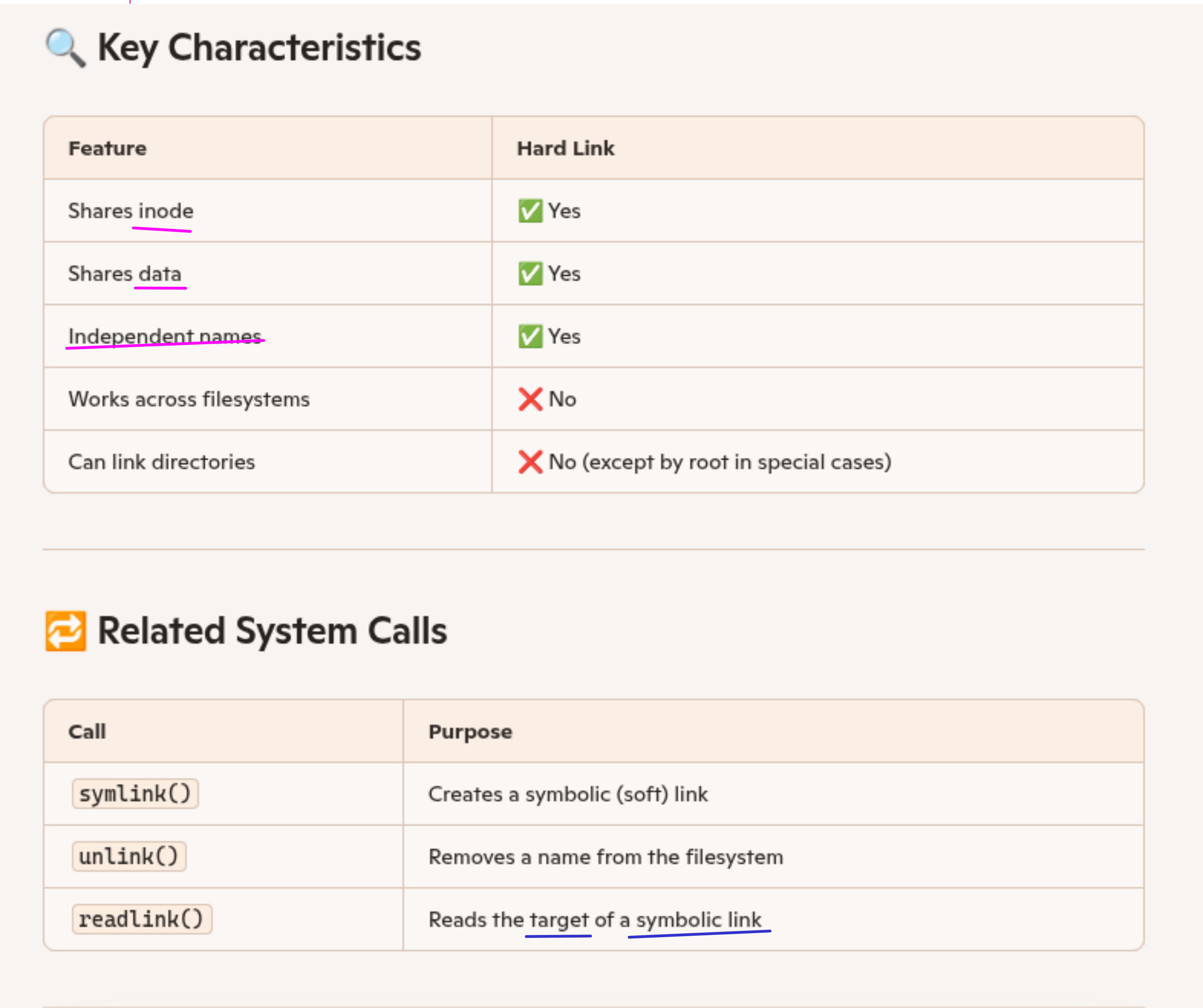
🧠 How It Works

- Both `oldpath` and `newpath` refer to the **same inode** on disk
- The file is not deleted until **all hard links** are removed and no process has it open
- Works only within the **same filesystem**

🧪 Example

```c
link("original.txt", "alias.txt");
```

- Creates `alias.txt` as a hard link to `original.txt`
- Changes made to one are reflected in the other

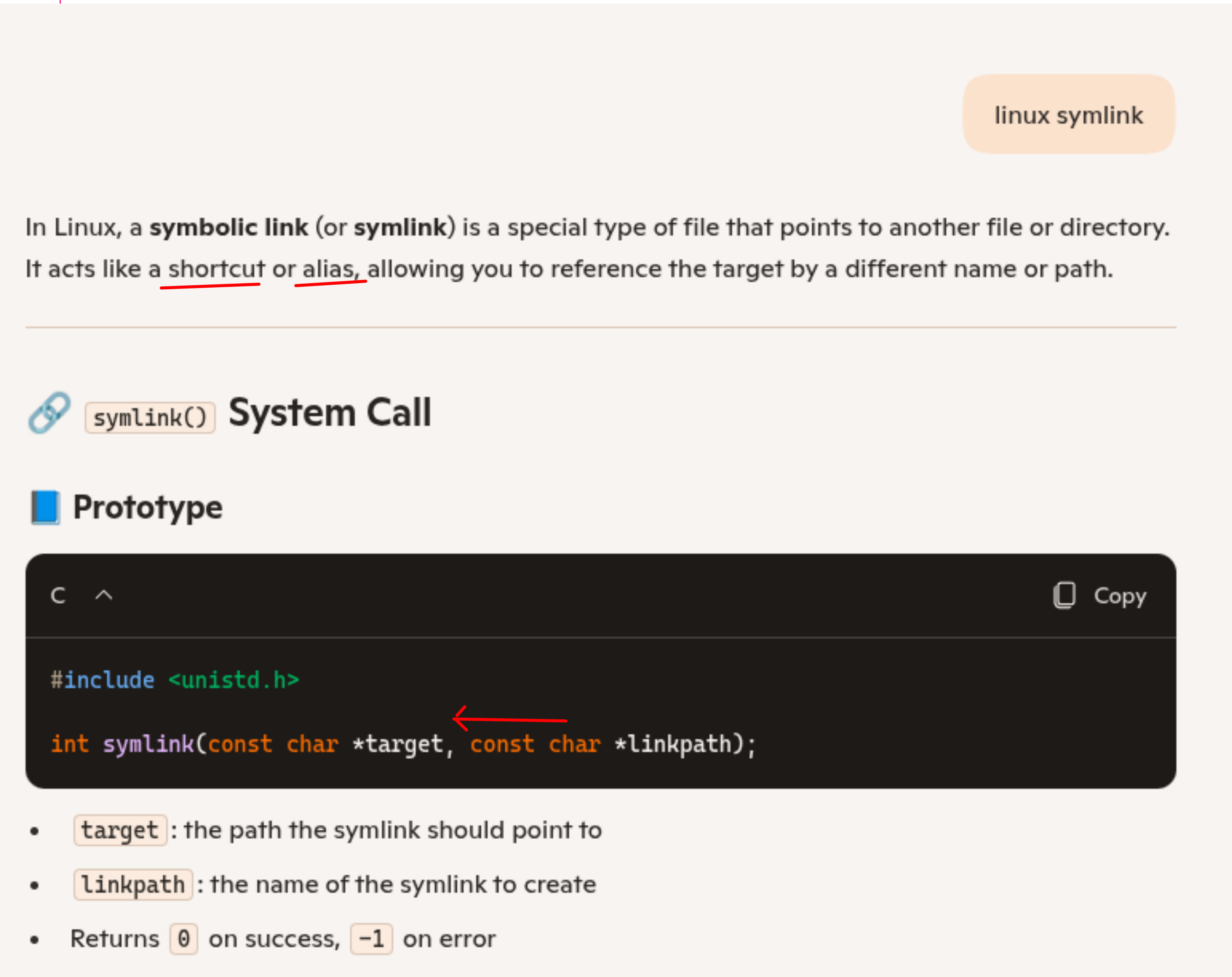


🔍 Key Characteristics

Feature	Hard Link
Shares inode	✅ Yes
Shares data	✅ Yes
Independent names	✅ Yes
Works across filesystems	❌ No
Can link directories	❌ No (except by root in special cases)

🔁 Related System Calls

Call	Purpose
`symlink()`	Creates a symbolic (soft) link
`unlink()`	Removes a name from the filesystem
`readlink()`	Reads the target of a symbolic link



linux symlink

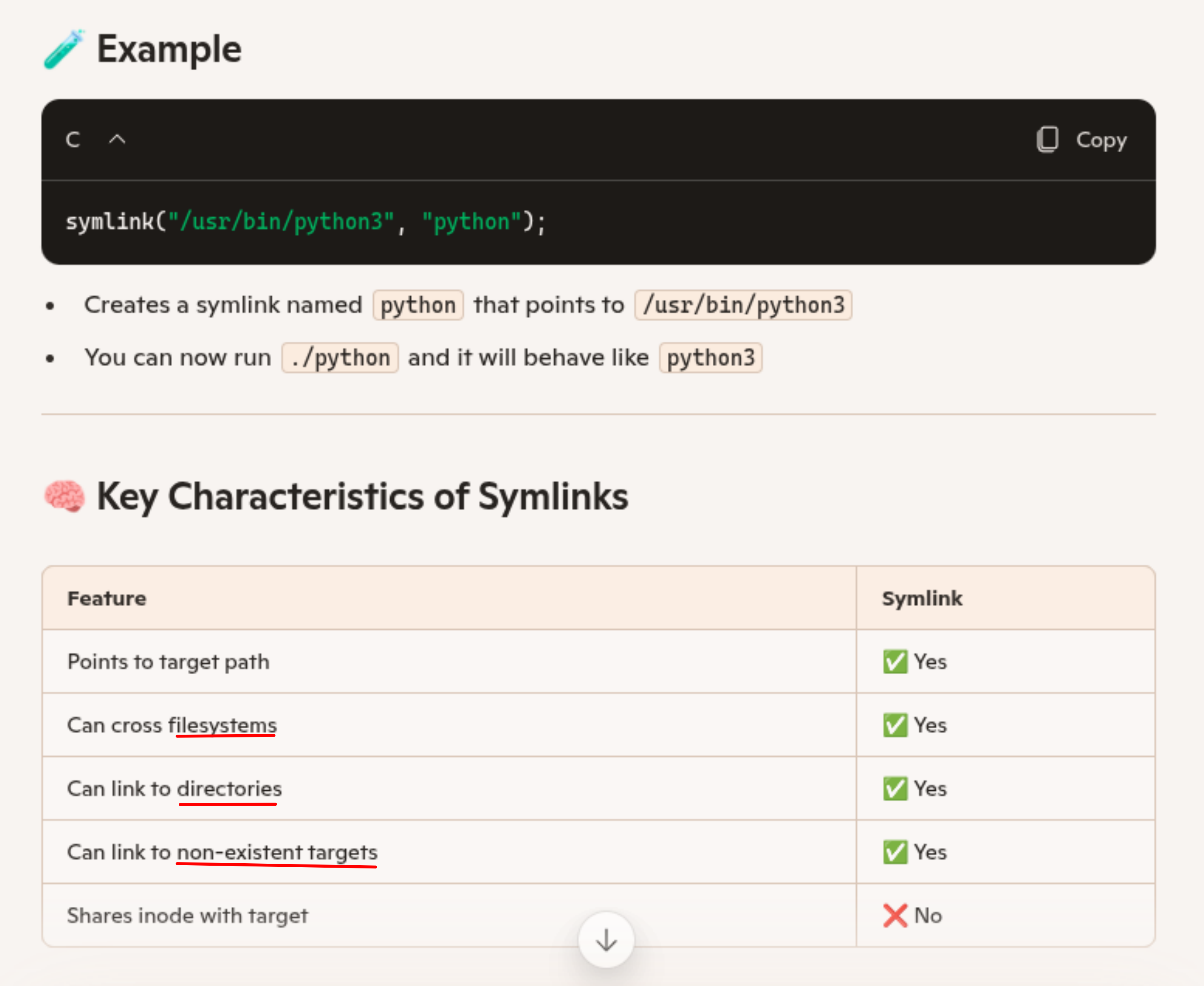
In Linux, a **symbolic link** (or **symlink**) is a special type of file that points to another file or directory. It acts like a shortcut or alias, allowing you to reference the target by a different name or path.

🔗 `symlink()` System Call

📘 Prototype

```c
#include <unistd.h>

int symlink(const char *target, const char *linkpath);
```

- `target`: the path the symlink should point to
- `linkpath`: the name of the symlink to create
- Returns `0` on success, `-1` on error



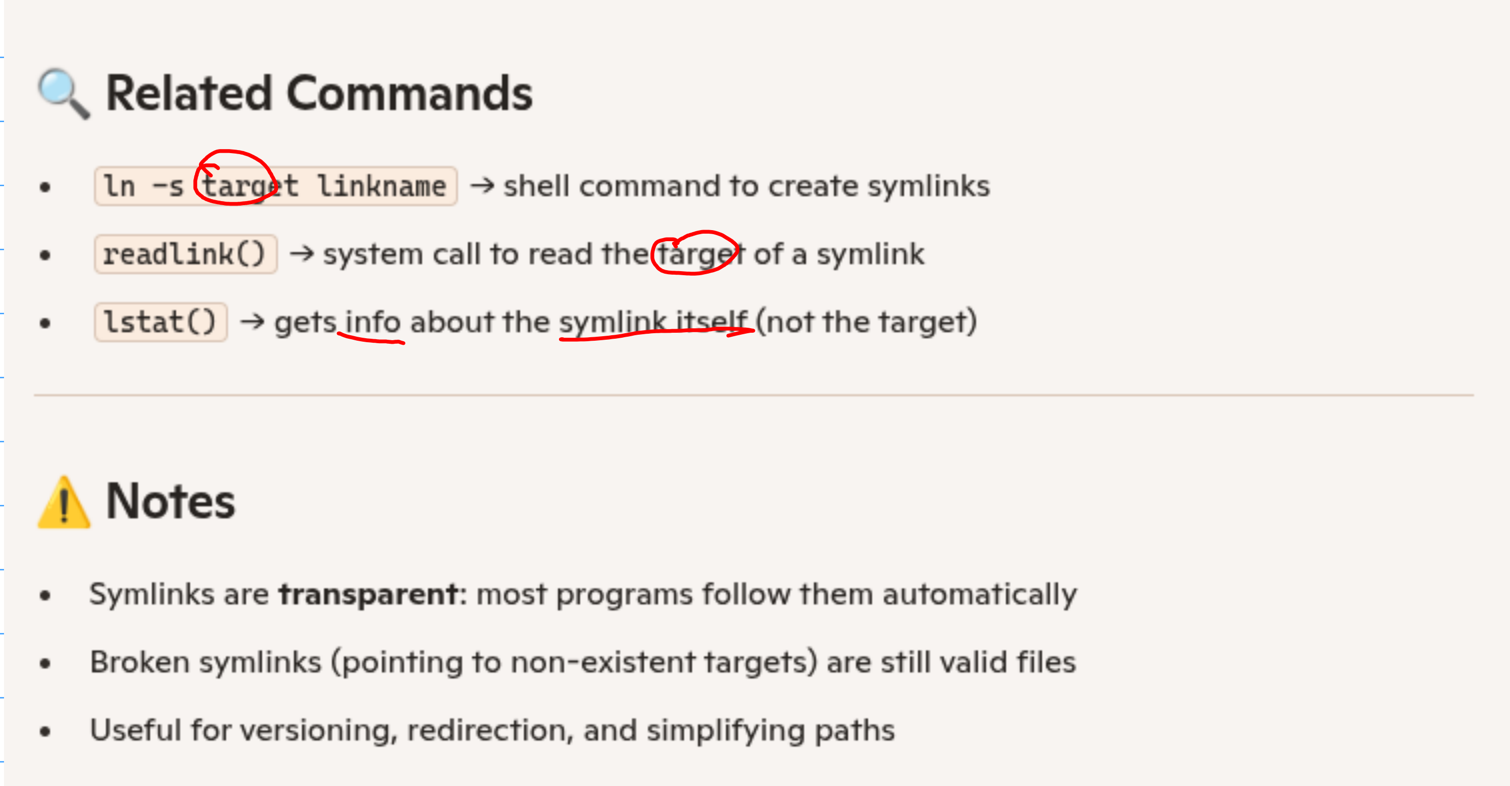
🧪 Example

```c
symlink("/usr/bin/python3", "python");
```

- Creates a symlink named `python` that points to `/usr/bin/python3`
- You can now run `./python` and it will behave like `python3`

🧠 Key Characteristics of Symlinks

Feature	**Symlink**
Points to target path	✅ Yes
Can cross filesystems	✅ Yes
Can link to directories	✅ Yes
Can link to non-existent targets	✅ Yes
Shares inode with target	❌ No

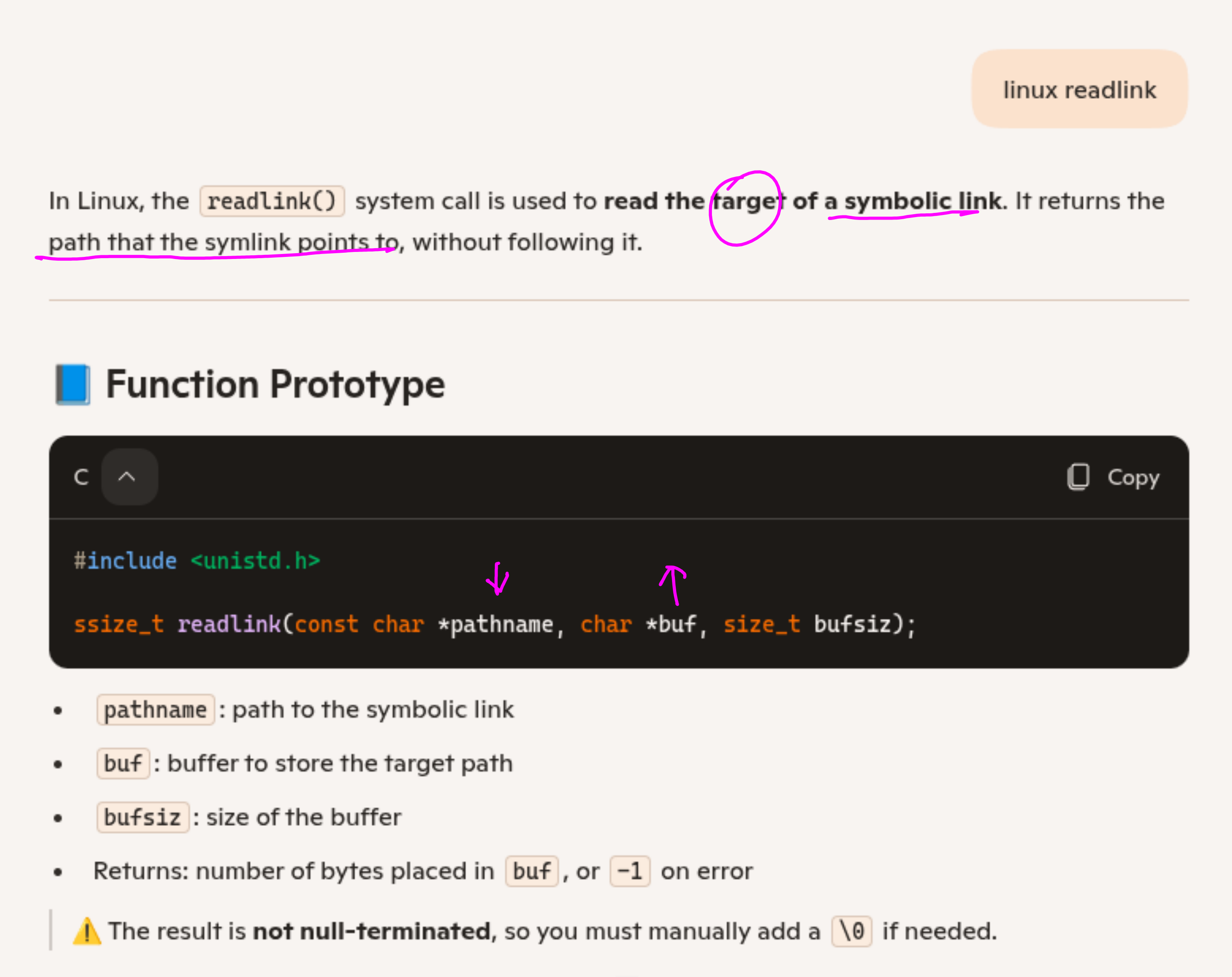


🔍 Related Commands

- `ln -s target linkname` → shell command to create symlinks
- `readlink()` → system call to read the target of a symlink
- `lstat()` → gets info about the symlink itself (not the target)

⚠️ Notes

- Symlinks are **transparent**: most programs follow them automatically
- Broken symlinks (pointing to non-existent targets) are still valid files
- Useful for versioning, redirection, and simplifying paths



linux readlink

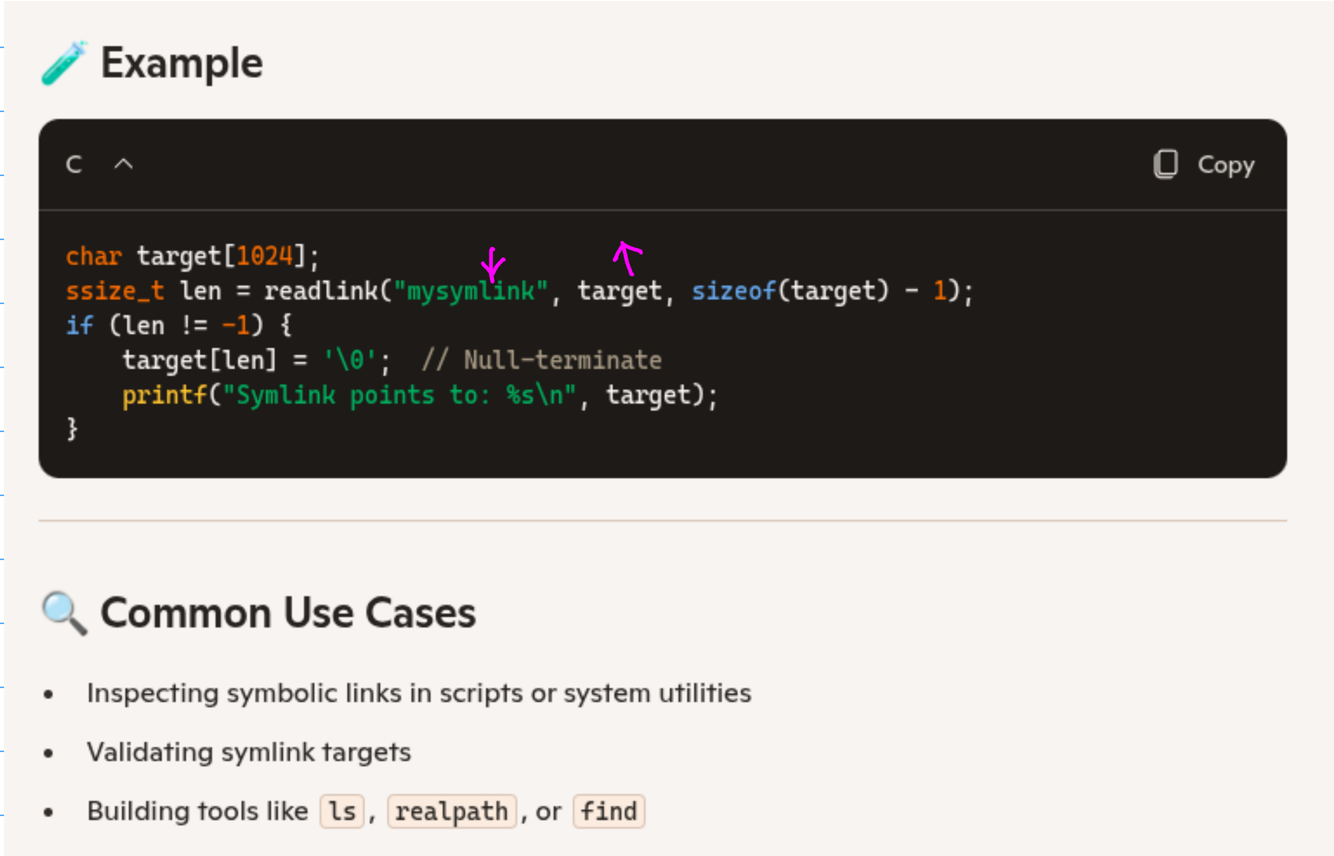
In Linux, the `readlink()` system call is used to **read the target of a symbolic link**. It returns the path that the symlink points to, without following it.

📘 Function Prototype

```c
#include <unistd.h>

ssize_t readlink(const char *pathname, char *buf, size_t bufsiz);
```

- `pathname`: path to the symbolic link
- `buf`: buffer to store the target path
- `bufsiz`: size of the buffer
- Returns: number of bytes placed in `buf`, or `-1` on error

> ⚠️ The result is **not null-terminated**, so you must manually add a `\0` if needed.

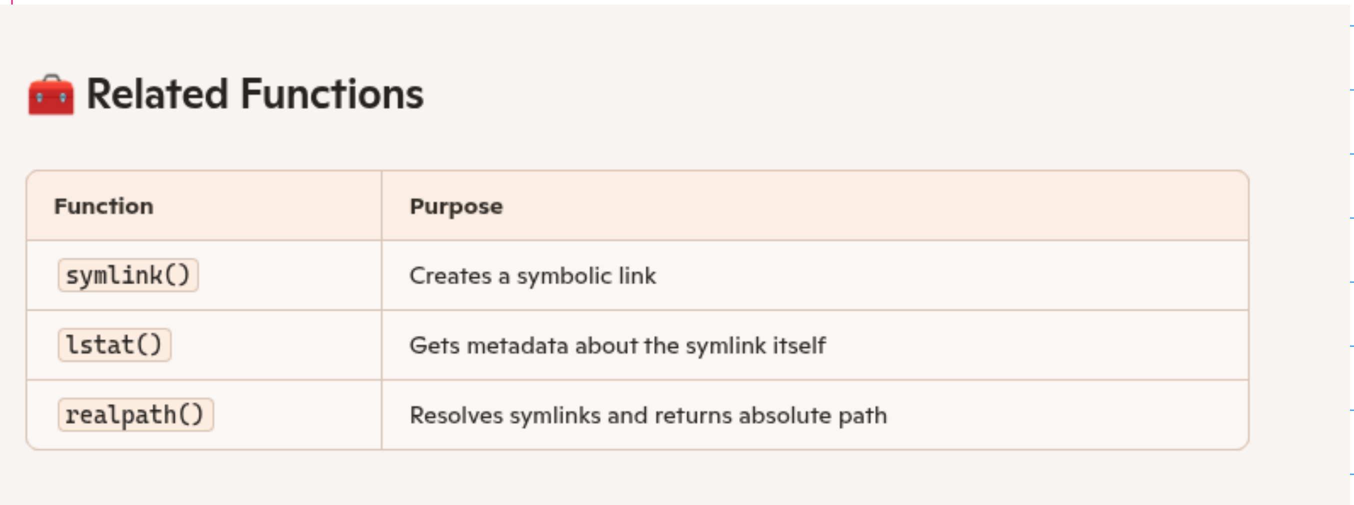


🧪 Example

```c
char target[1024];
ssize_t len = readlink("mysymlink", target, sizeof(target) - 1);
if (len != -1) {
    target[len] = '\0';  // Null-terminate
    printf("Symlink points to: %s\n", target);
}
```

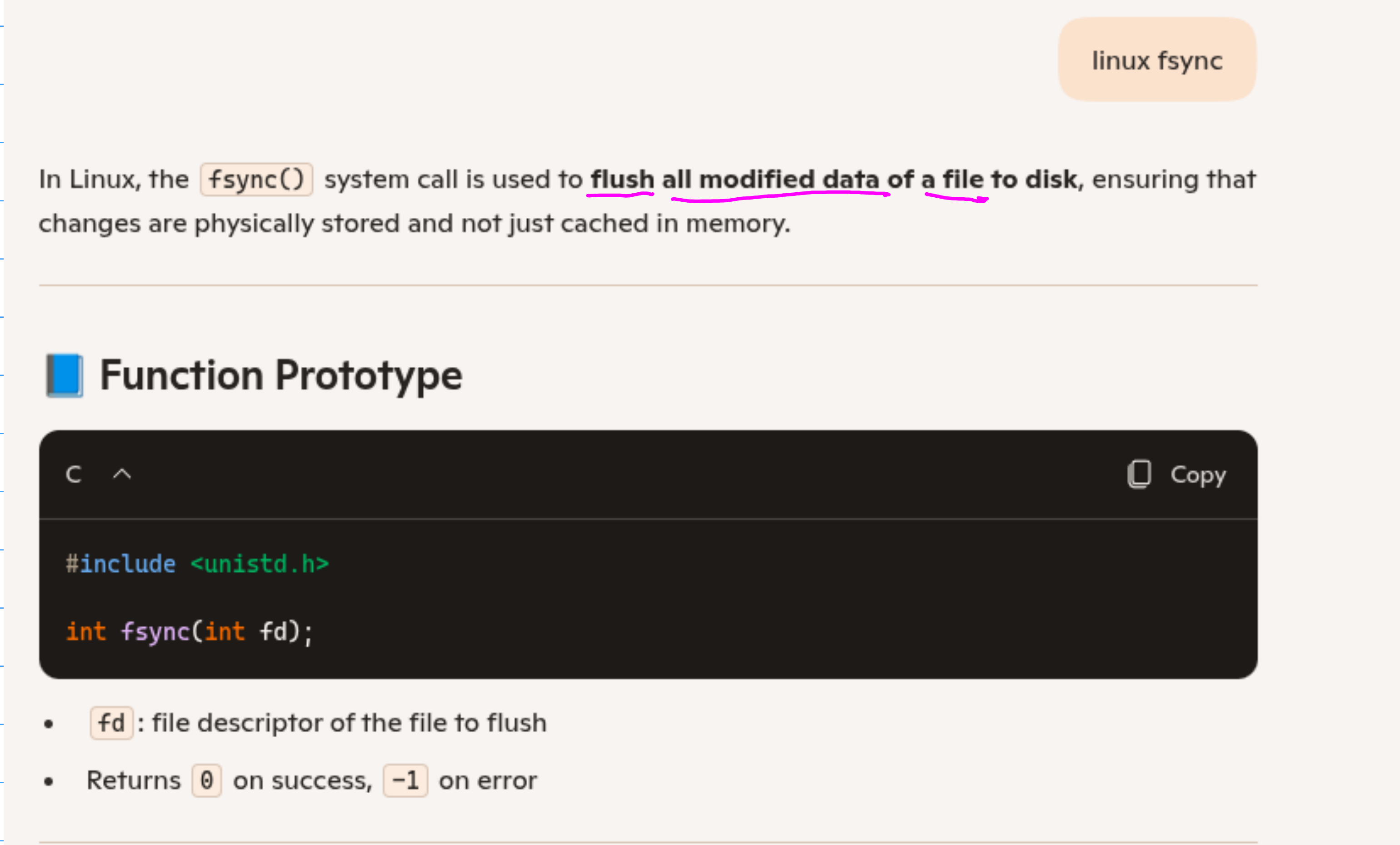
🔍 Common Use Cases

- Inspecting symbolic links in scripts or system utilities
- Validating symlink targets
- Building tools like `ls`, `realpath`, or `find`



🧰 Related Functions

Function	Purpose
`symlink()`	Creates a symbolic link
`lstat()`	Gets metadata about the symlink itself
`realpath()`	Resolves symlinks and returns absolute path



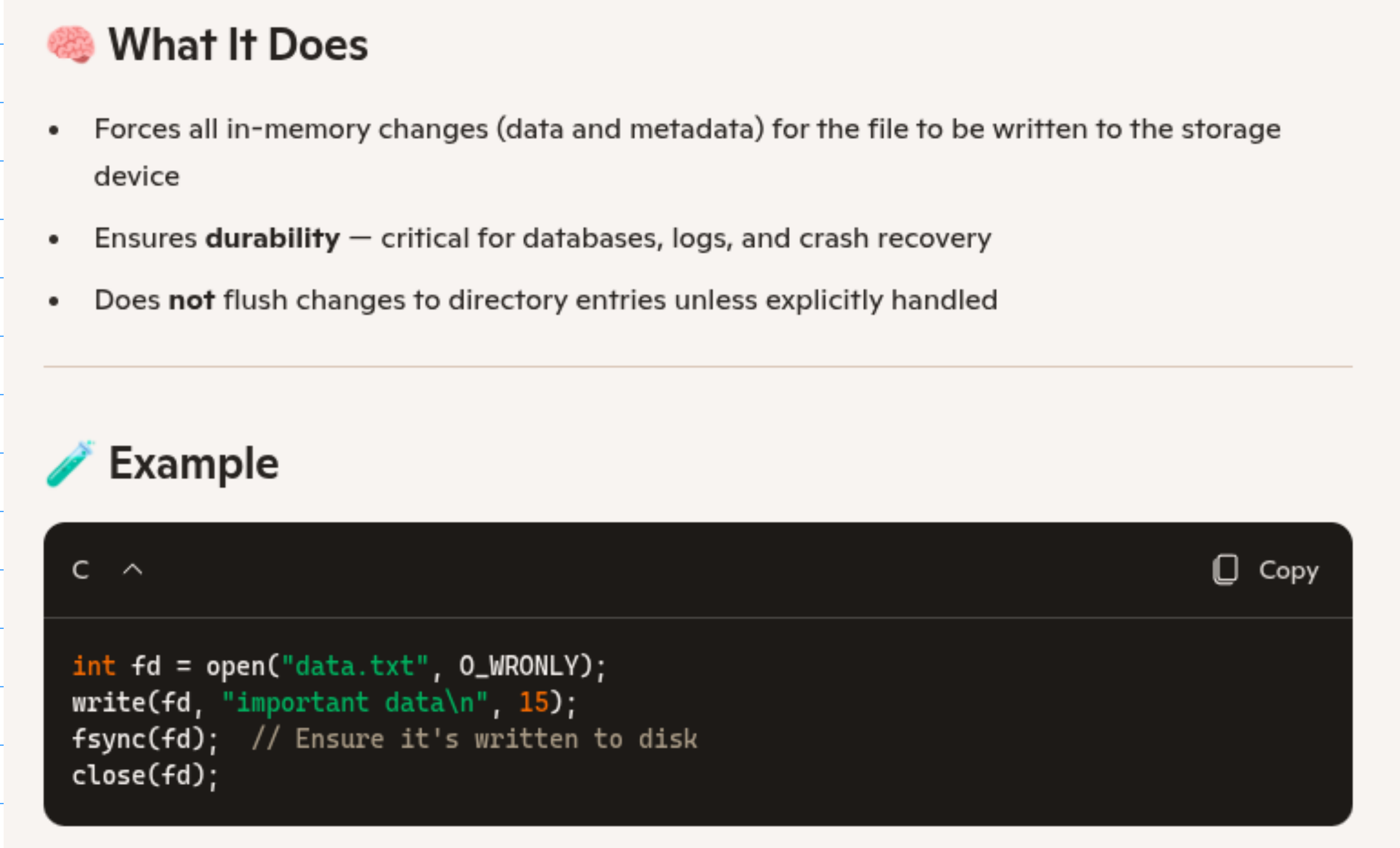
linux fsync

In Linux, the `fsync()` system call is used to **flush all modified data of a file to disk**, ensuring that changes are physically stored and not just cached in memory.

📘 Function Prototype

```c
#include <unistd.h>

int fsync(int fd);
```

- `fd` : file descriptor of the file to flush
- Returns `0` on success, `-1` on error



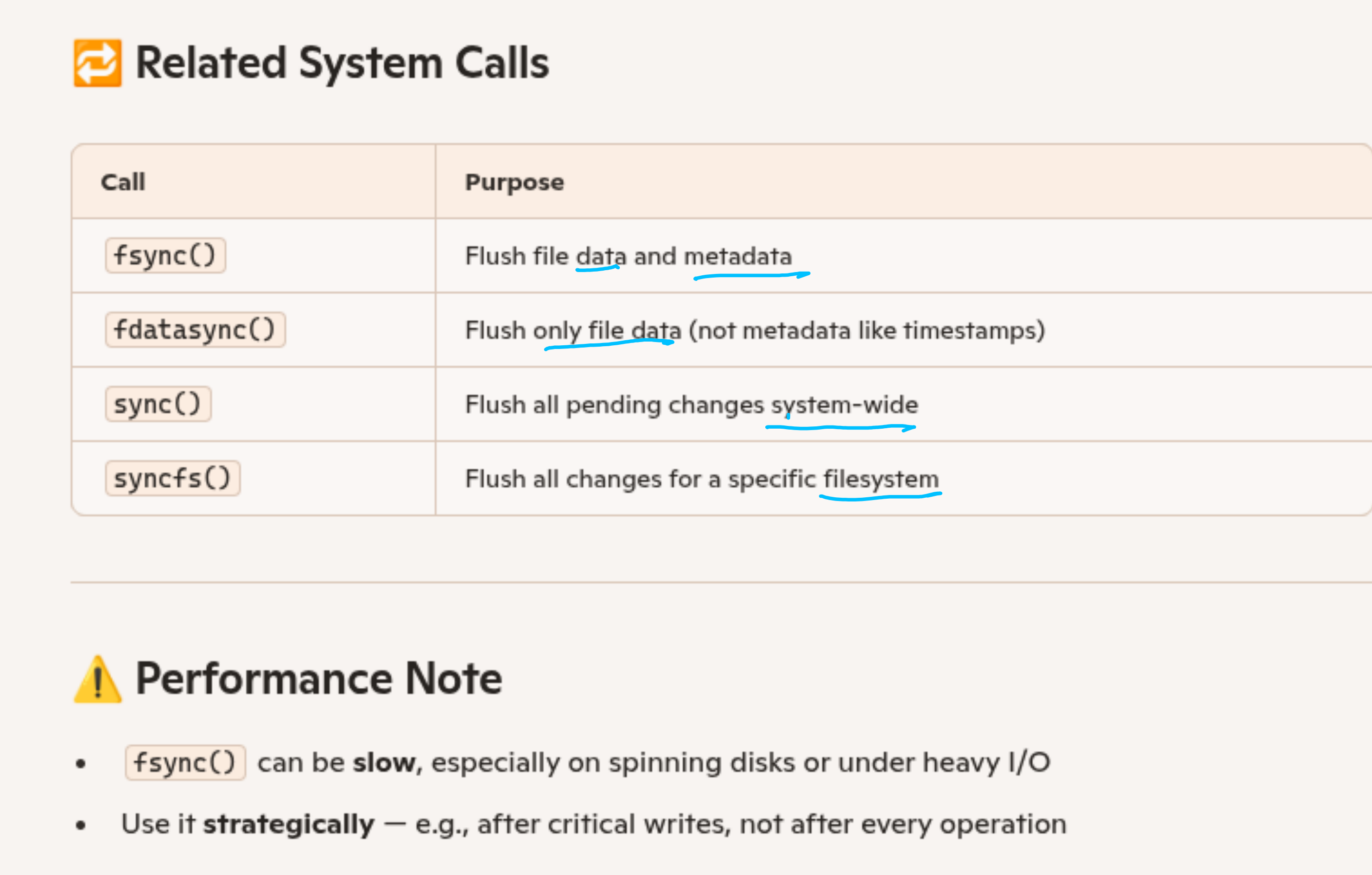
🧠 What It Does

- Forces all in-memory changes (data and metadata) for the file to be written to the storage device
- Ensures **durability** — critical for databases, logs, and crash recovery
- Does **not** flush changes to directory entries unless explicitly handled

---

🧪 Example

```c
int fd = open("data.txt", O_WRONLY);
write(fd, "important data\n", 15);
fsync(fd);  // Ensure it's written to disk
close(fd);
```

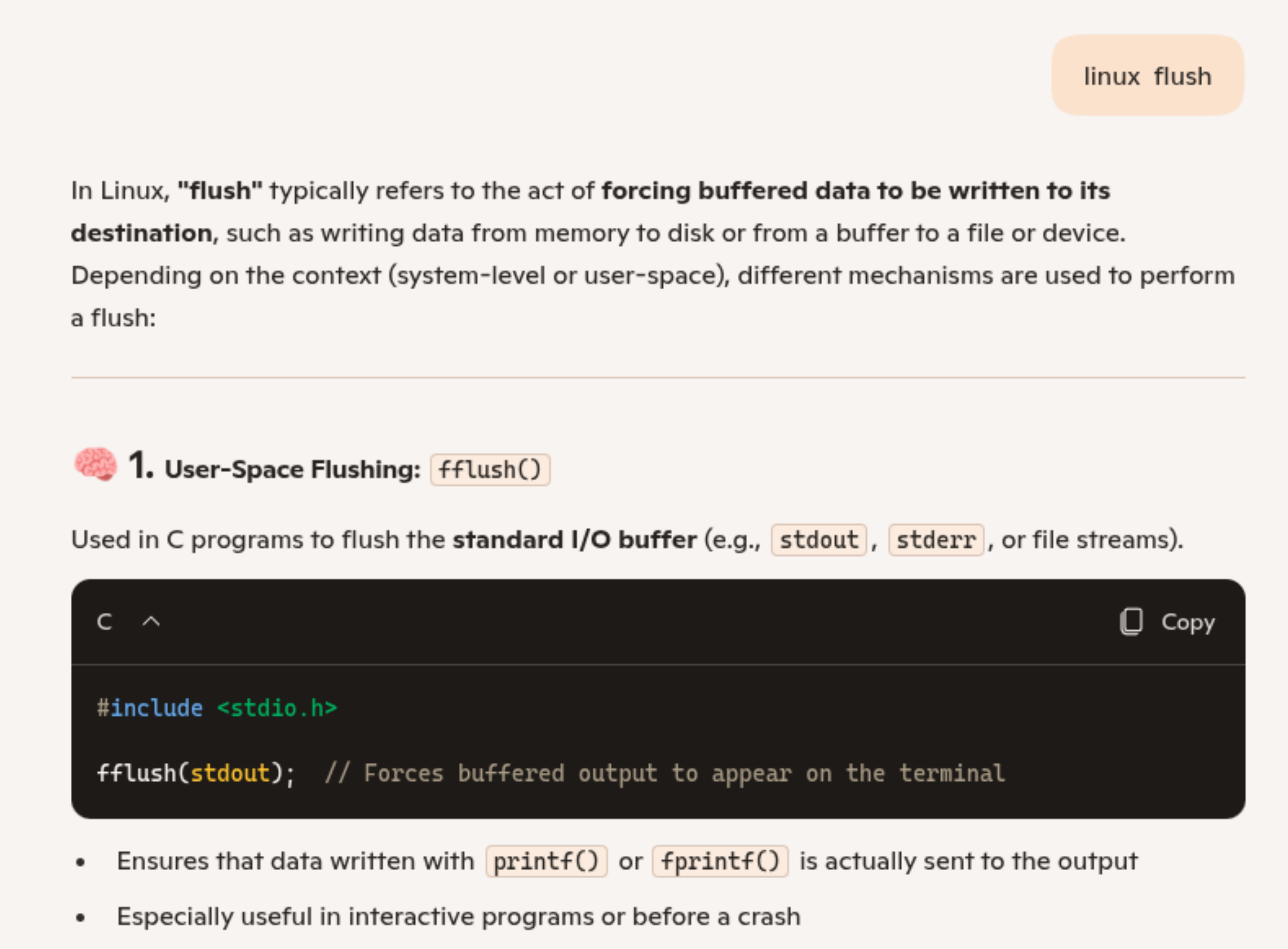


🔁 Related System Calls

Call	Purpose
`fsync()`	Flush file data and metadata
`fdatasync()`	Flush only file data (not metadata like timestamps)
`sync()`	Flush all pending changes system-wide
`syncfs()`	Flush all changes for a specific filesystem

⚠️ Performance Note

- `fsync()` can be **slow**, especially on spinning disks or under heavy I/O
- Use it **strategically** — e.g., after critical writes, not after every operation



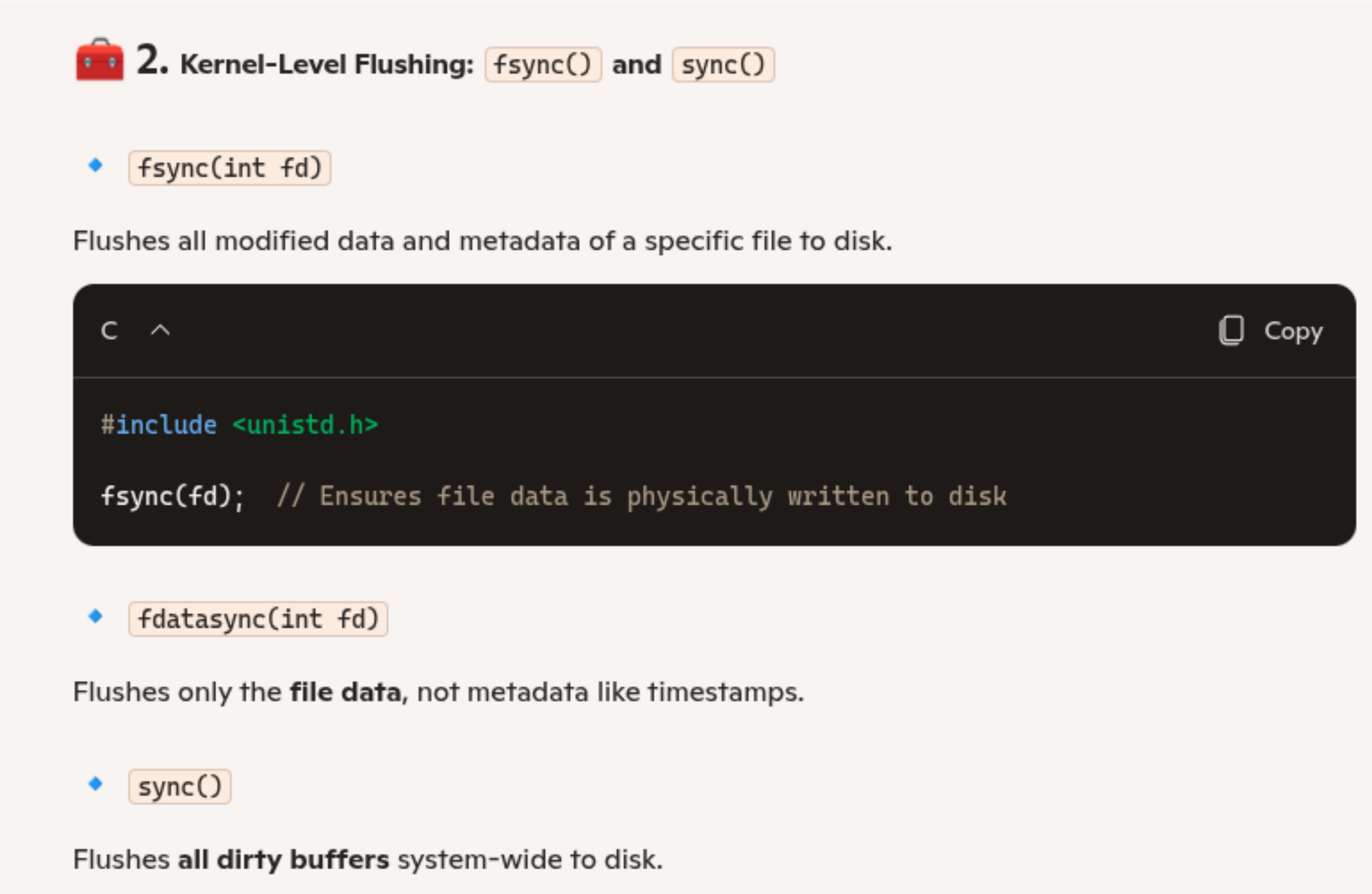
In Linux, **"flush"** typically refers to the act of **forcing buffered data to be written to its destination**, such as writing data from memory to disk or from a buffer to a file or device. Depending on the context (system-level or user-space), different mechanisms are used to perform a flush:

🧠 1. User-Space Flushing: `fflush()`

Used in C programs to flush the **standard I/O buffer** (e.g., `stdout`, `stderr`, or file streams).

```c
#include <stdio.h>

fflush(stdout);  // Forces buffered output to appear on the terminal
```

- Ensures that data written with `printf()` or `fprintf()` is actually sent to the output
- Especially useful in interactive programs or before a crash



🧰 2. Kernel-Level Flushing: `fsync()` and `sync()`

🔹 `fsync(int fd)`

Flushes all modified data and metadata of a specific file to disk.

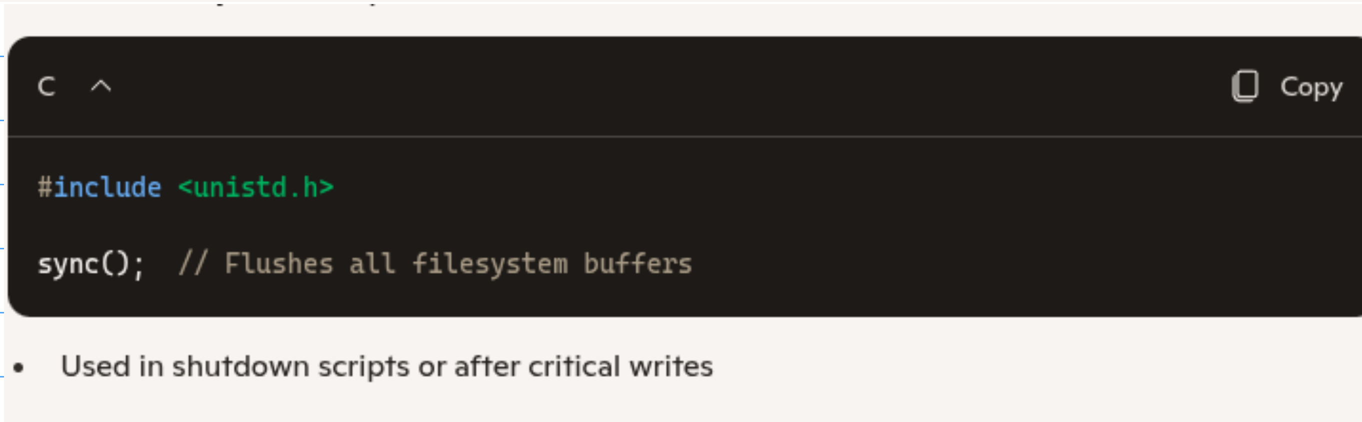
```c
#include <unistd.h>

fsync(fd);  // Ensures file data is physically written to disk
```

🔹 `fdatasync(int fd)`

Flushes only the **file data**, not metadata like timestamps.

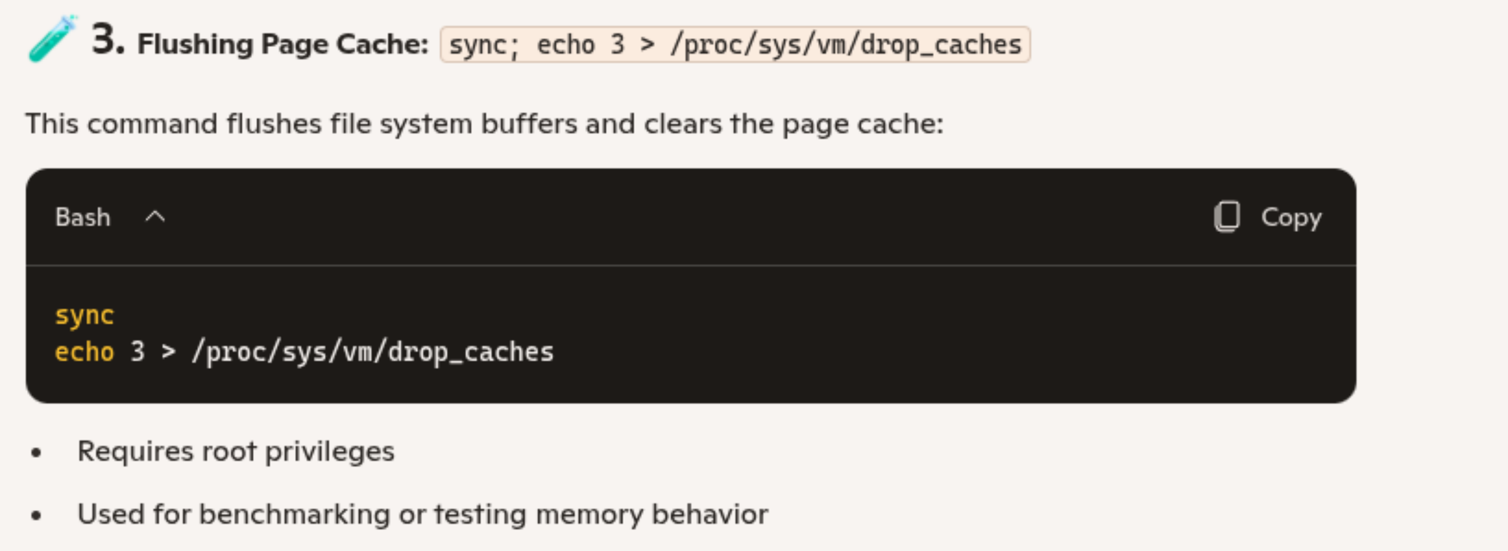
🔹 `sync()`

Flushes **all dirty buffers** system-wide to disk.



```c
#include <unistd.h>

sync();  // Flushes all filesystem buffers
```

- Used in shutdown scripts or after critical writes

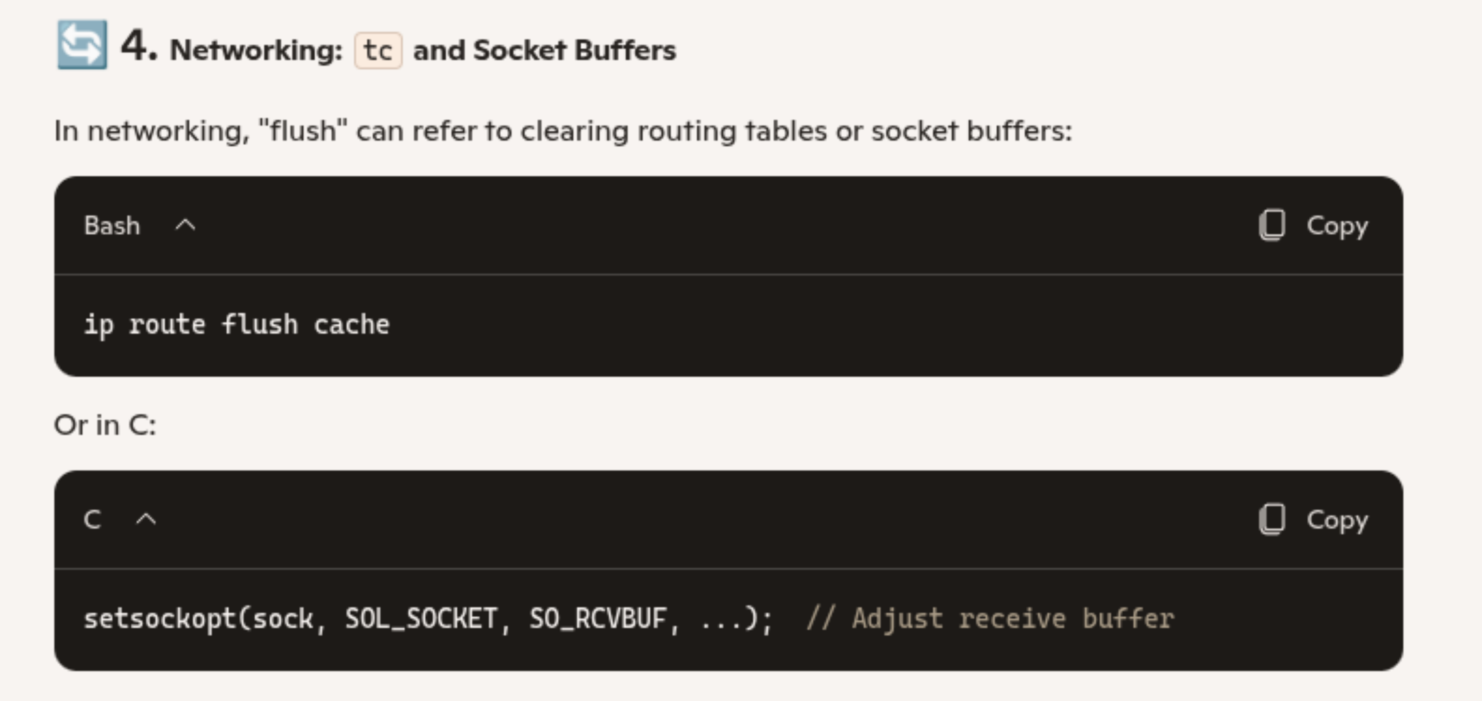


3. Flushing Page Cache: `sync; echo 3 > /proc/sys/vm/drop_caches`

This command flushes file system buffers and clears the page cache:

```bash
sync
echo 3 > /proc/sys/vm/drop_caches
```

- Requires root privileges
- Used for benchmarking or testing memory behavior



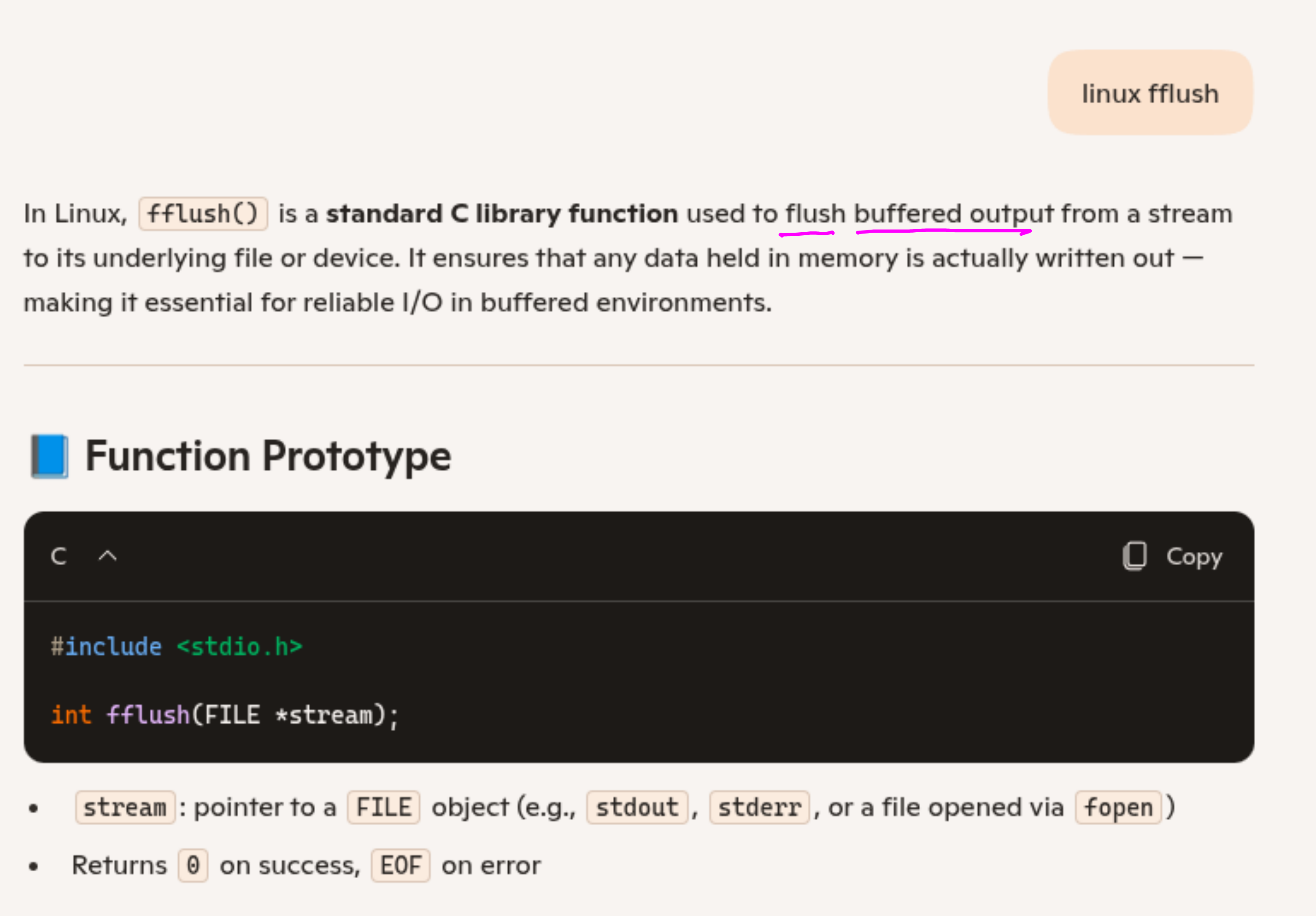
🔄 4. Networking: `tc` and Socket Buffers

In networking, "flush" can refer to clearing routing tables or socket buffers:

```bash
ip route flush cache
```

Or in C:

```c
setsockopt(sock, SOL_SOCKET, SO_RCVBUF, ...);  // Adjust receive buffer
```



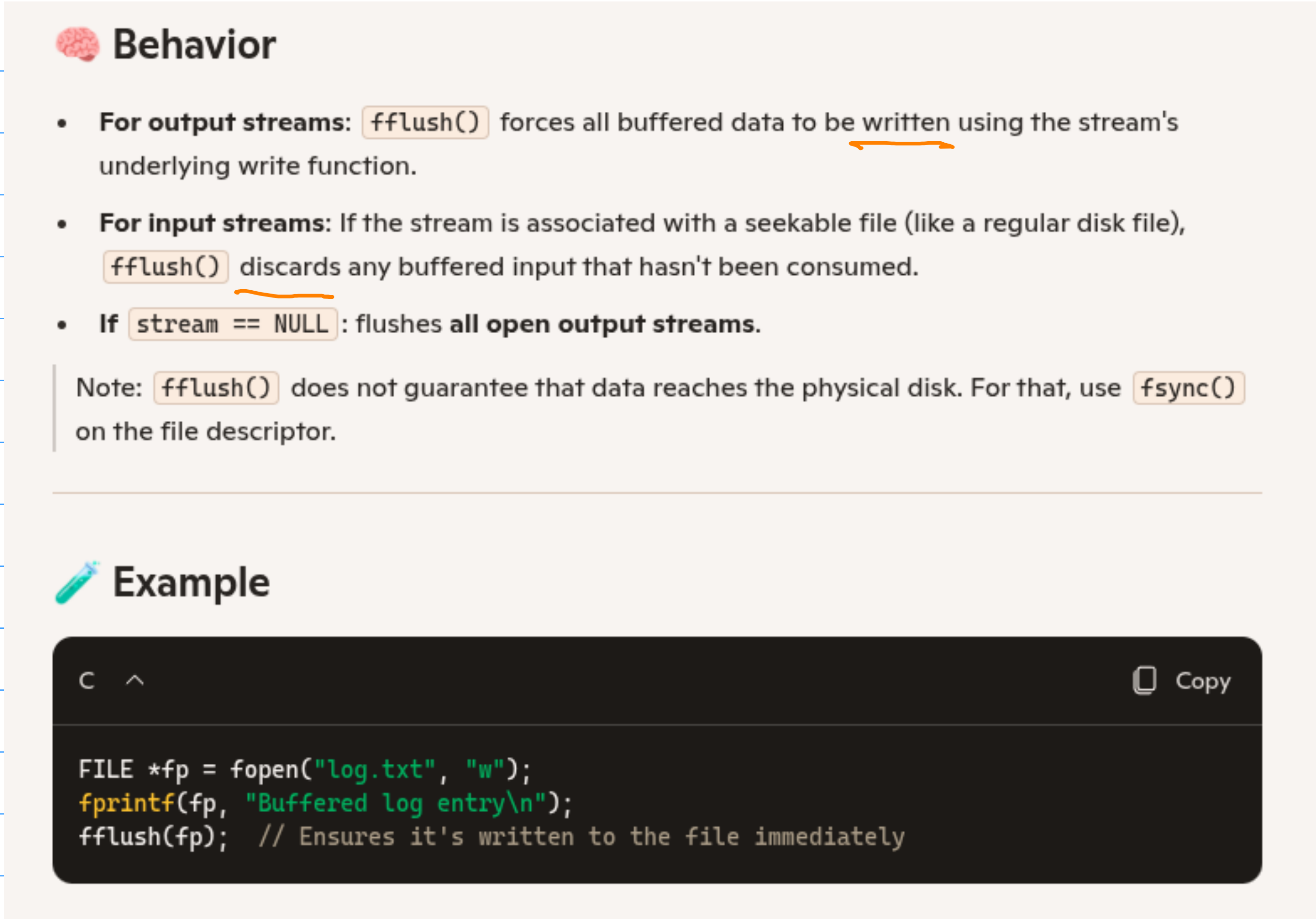
linux fflush

In Linux, `fflush()` is a **standard C library function** used to flush buffered output from a stream to its underlying file or device. It ensures that any data held in memory is actually written out — making it essential for reliable I/O in buffered environments.

📘 Function Prototype

```c
#include <stdio.h>

int fflush(FILE *stream);
```

- `stream`: pointer to a `FILE` object (e.g., `stdout`, `stderr`, or a file opened via `fopen`)
- Returns `0` on success, `EOF` on error



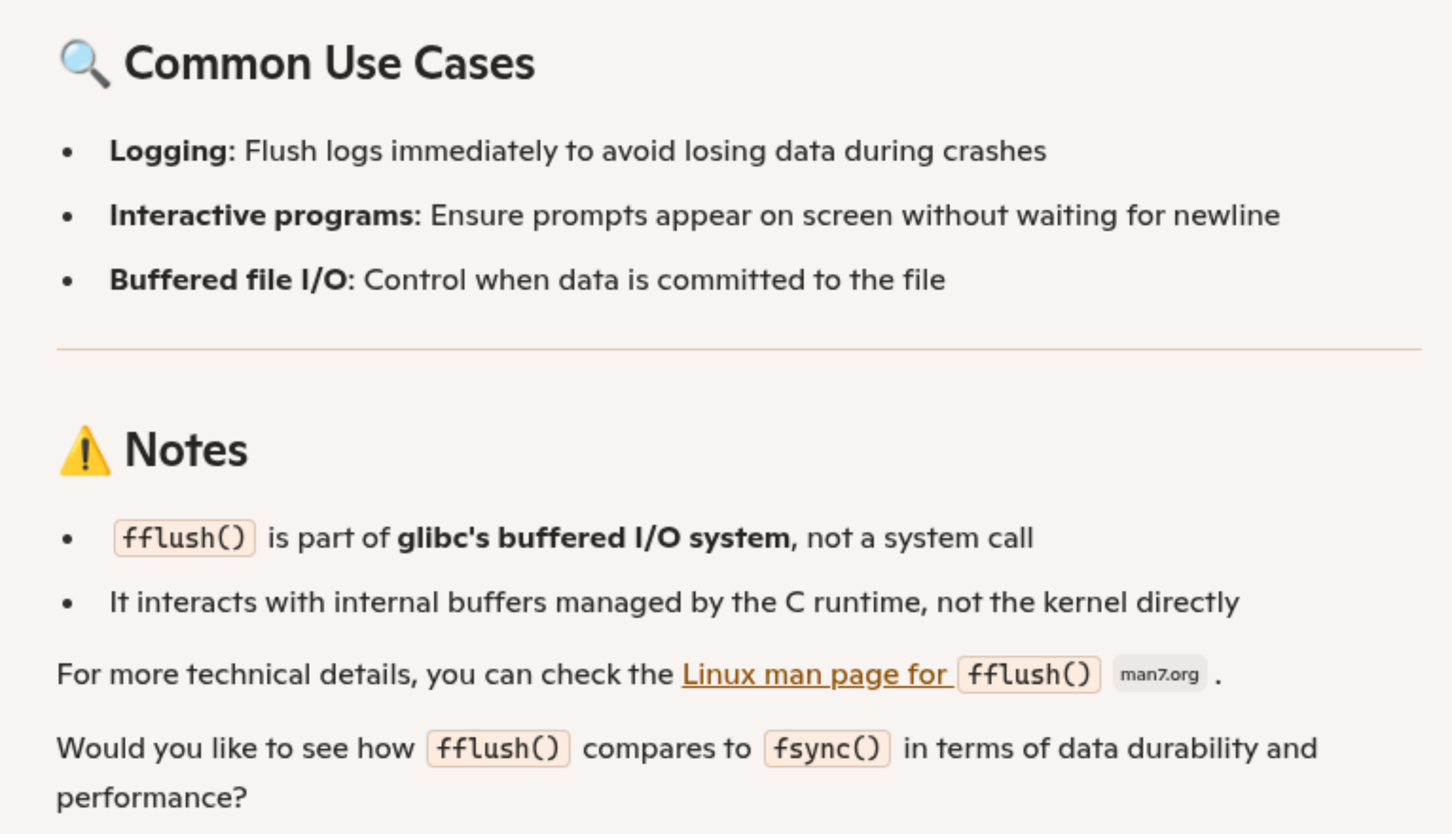
🧠 Behavior

- **For output streams:** `fflush()` forces all buffered data to be written using the stream's underlying write function.
- **For input streams:** If the stream is associated with a seekable file (like a regular disk file), `fflush()` discards any buffered input that hasn't been consumed.
- **If** `stream == NULL`: flushes **all open output streams**.

> Note: `fflush()` does not guarantee that data reaches the physical disk. For that, use `fsync()` on the file descriptor.

🧪 Example

```c
FILE *fp = fopen("log.txt", "w");
fprintf(fp, "Buffered log entry\n");
fflush(fp);  // Ensures it's written to the file immediately
```



🔍 Common Use Cases

- **Logging**: Flush logs immediately to avoid losing data during crashes
- **Interactive programs**: Ensure prompts appear on screen without waiting for newline
- **Buffered file I/O**: Control when data is committed to the file

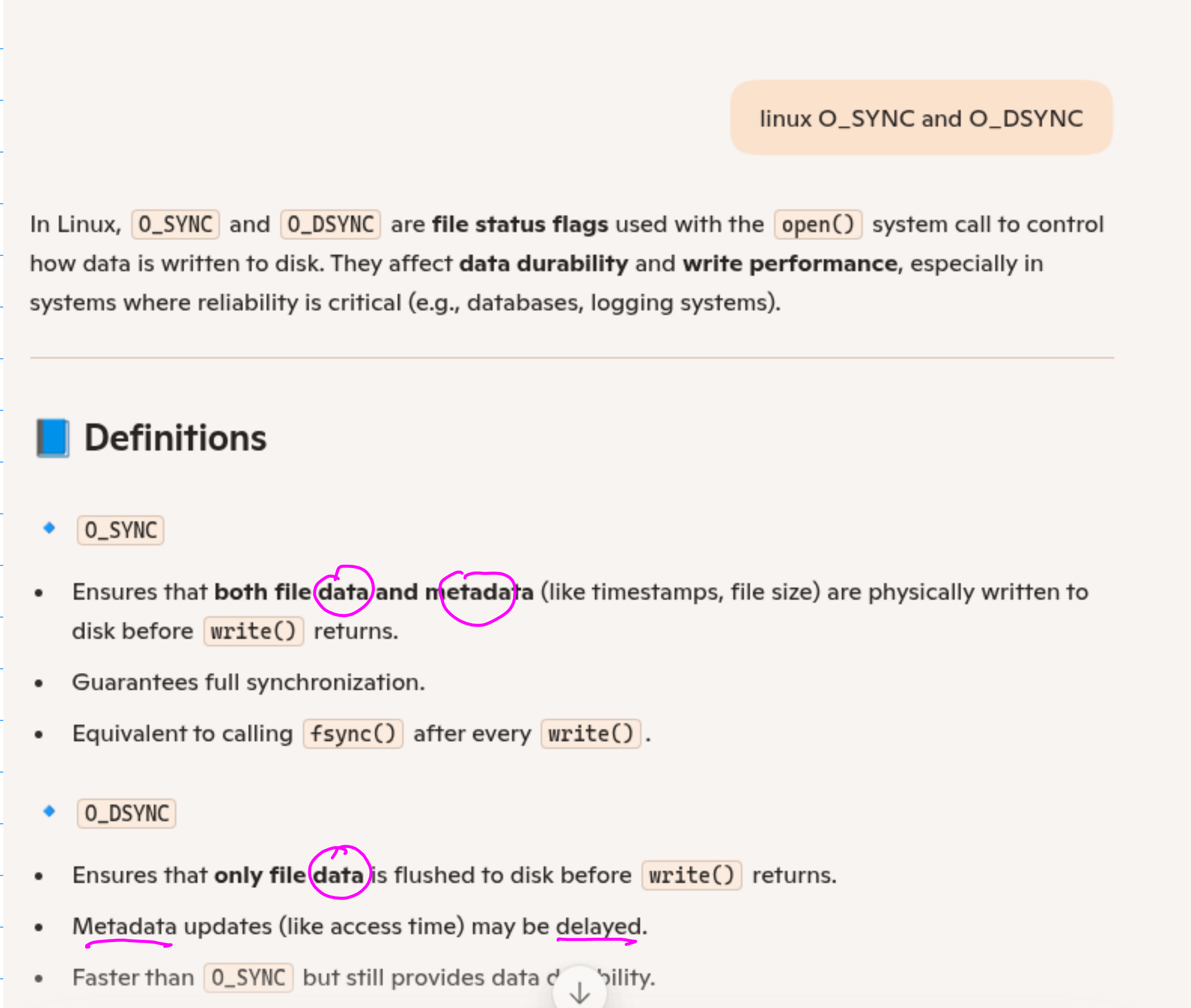
---

⚠️ Notes

- `fflush()` is part of **glibc's buffered I/O system**, not a system call
- It interacts with internal buffers managed by the C runtime, not the kernel directly

For more technical details, you can check the Linux man page for `fflush()` man7.org .

Would you like to see how `fflush()` compares to `fsync()` in terms of data durability and performance?



linux O_SYNC and O_DSYNC

In Linux, `O_SYNC` and `O_DSYNC` are **file status flags** used with the `open()` system call to control how data is written to disk. They affect **data durability** and **write performance**, especially in systems where reliability is critical (e.g., databases, logging systems).

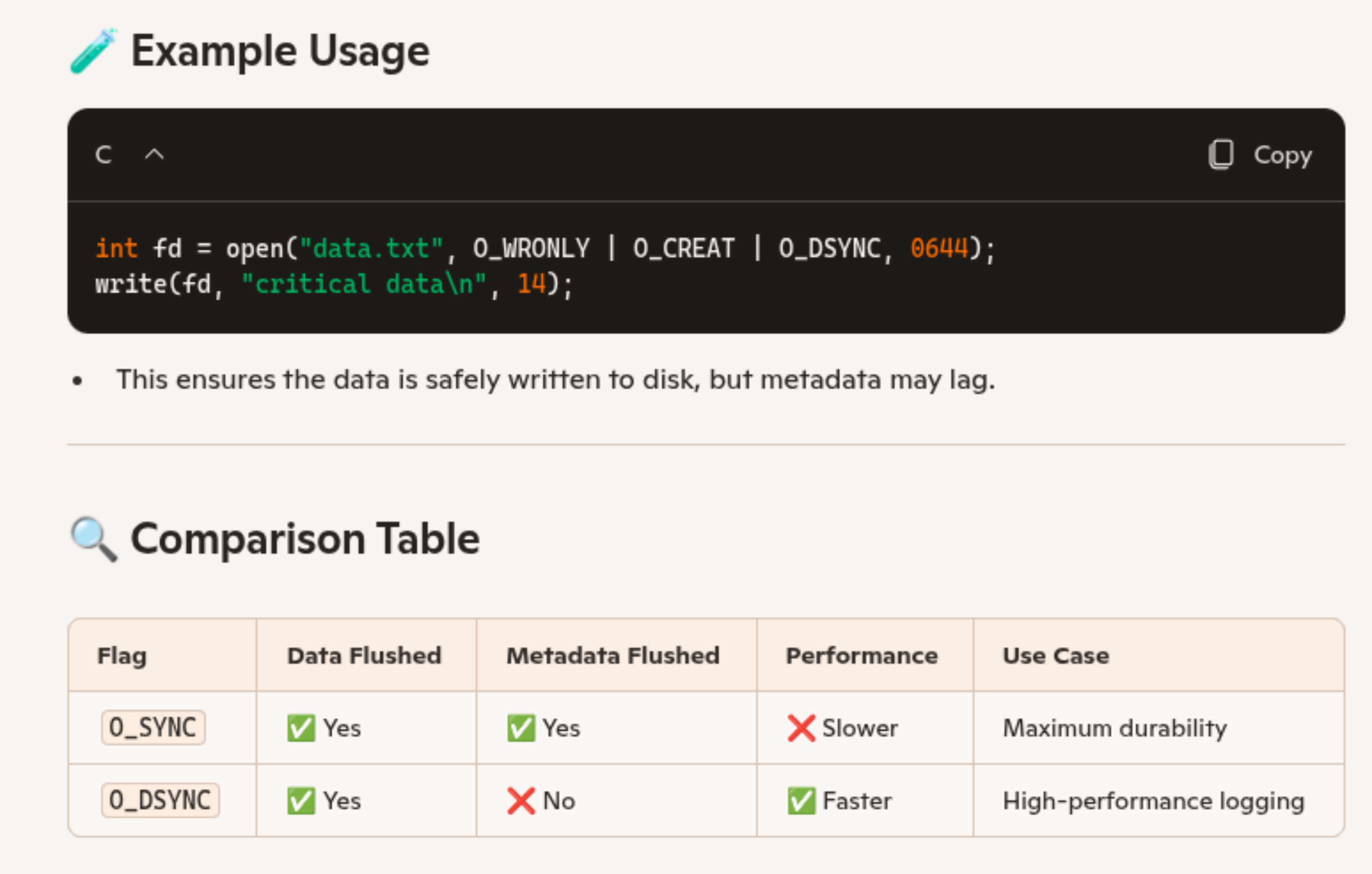
📘 Definitions

🔹 `O_SYNC`

- Ensures that **both file data and metadata** (like timestamps, file size) are physically written to disk before `write()` returns.
- Guarantees full synchronization.
- Equivalent to calling `fsync()` after every `write()`.

🔹 `O_DSYNC`

- Ensures that **only file data** is flushed to disk before `write()` returns.
- Metadata updates (like access time) may be delayed.
- Faster than `O_SYNC` but still provides data d bility.



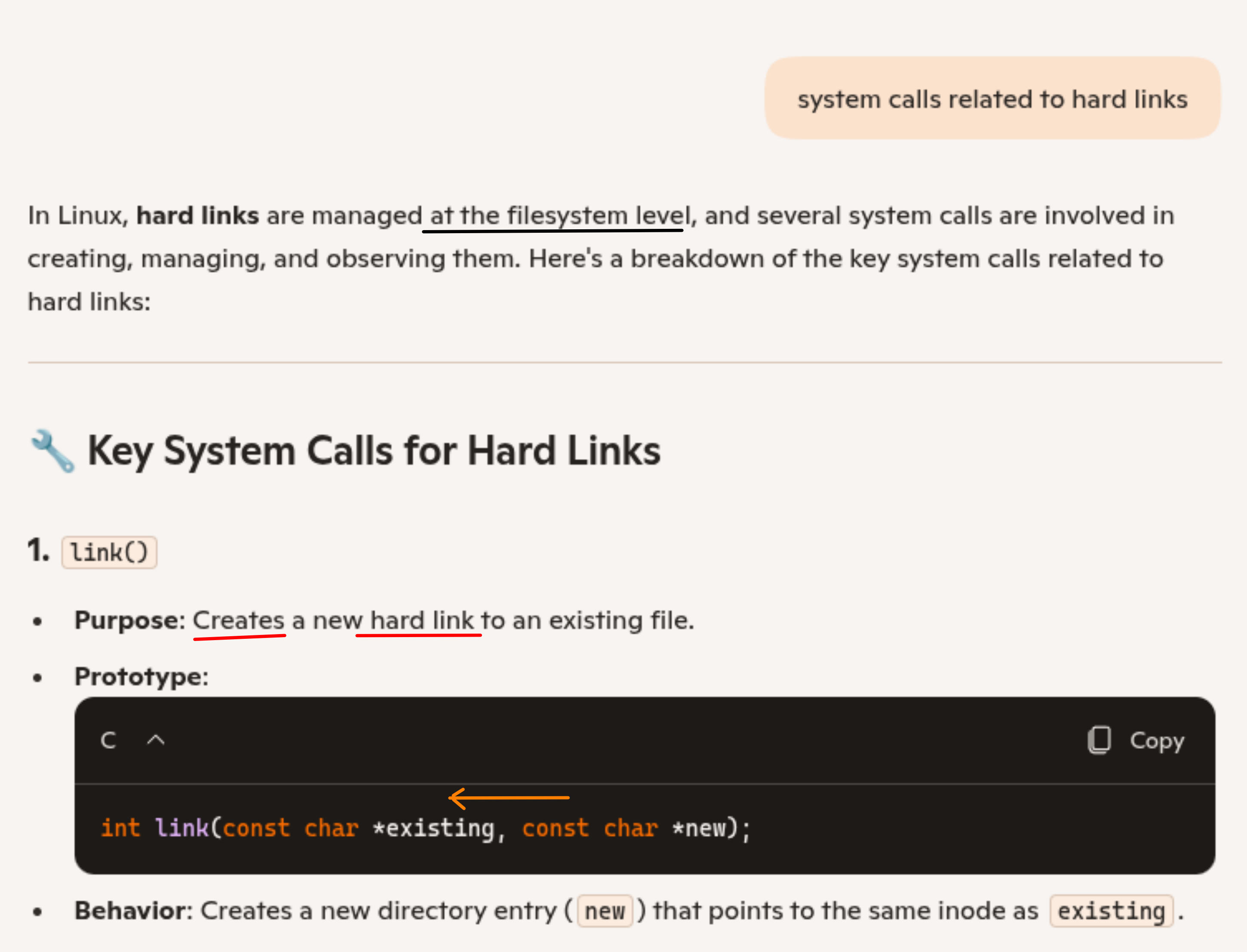
🧪 Example Usage

```c
int fd = open("data.txt", O_WRONLY | O_CREAT | O_DSYNC, 0644);
write(fd, "critical data\n", 14);
```

- This ensures the data is safely written to disk, but metadata may lag.

🔍 Comparison Table

Flag	Data Flushed	Metadata Flushed	Performance	Use Case
`O_SYNC`	✅ Yes	✅ Yes	❌ Slower	Maximum durability
`O_DSYNC`	✅ Yes	❌ No	✅ Faster	High-performance logging



system calls related to hard links

In Linux, **hard links** are managed at the filesystem level, and several system calls are involved in creating, managing, and observing them. Here's a breakdown of the key system calls related to hard links:

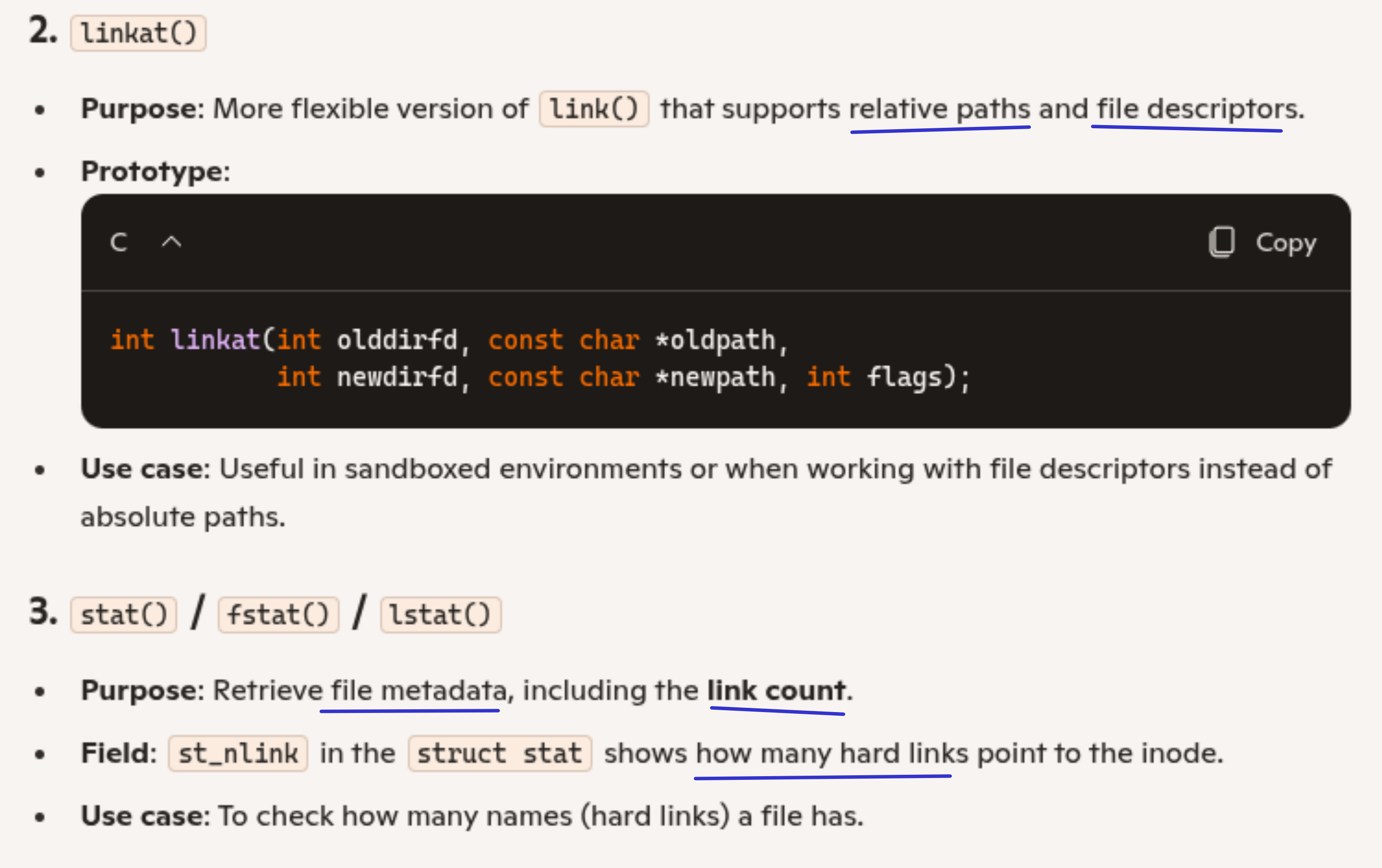
🔧 Key System Calls for Hard Links

1. `link()`

- **Purpose**: Creates a new hard link to an existing file.
- **Prototype**:

```c
int link(const char *existing, const char *new);
```

- **Behavior**: Creates a new directory entry (`new`) that points to the same inode as `existing`.



2. `linkat()`

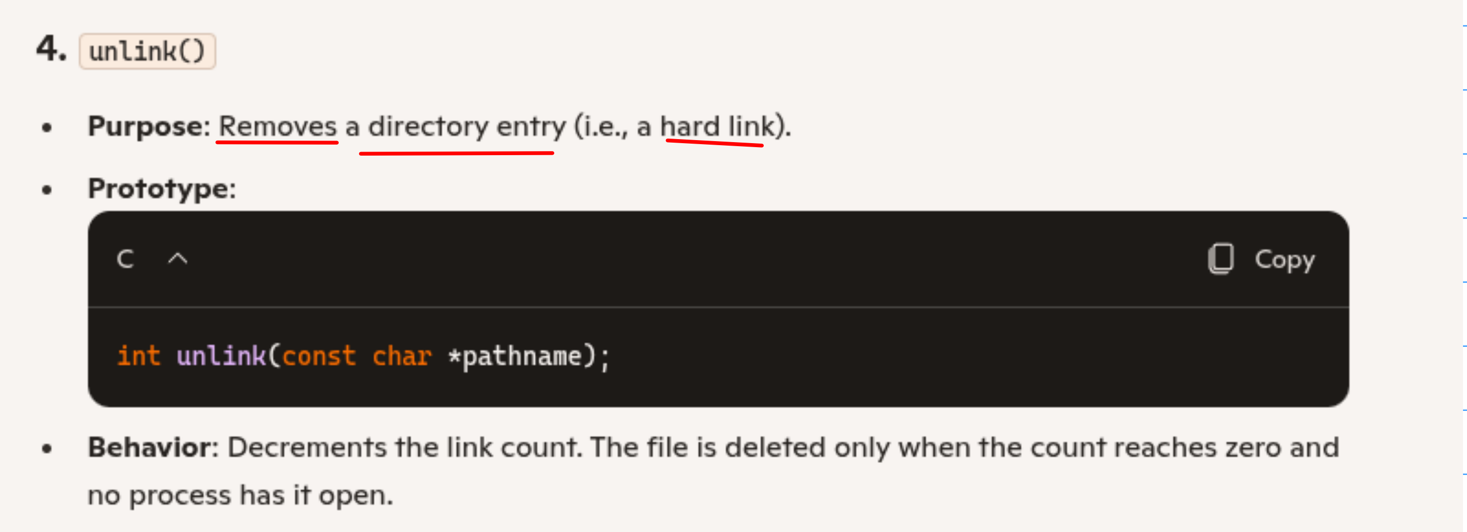
- **Purpose**: More flexible version of `link()` that supports relative paths and file descriptors.
- **Prototype**:

```c
int linkat(int olddirfd, const char *oldpath,
           int newdirfd, const char *newpath, int flags);
```

- **Use case:** Useful in sandboxed environments or when working with file descriptors instead of absolute paths.

3. `stat()` / `fstat()` / `lstat()`

- **Purpose**: Retrieve file metadata, including the **link count**.
- **Field:** `st_nlink` in the `struct stat` shows how many hard links point to the inode.
- **Use case:** To check how many names (hard links) a file has.

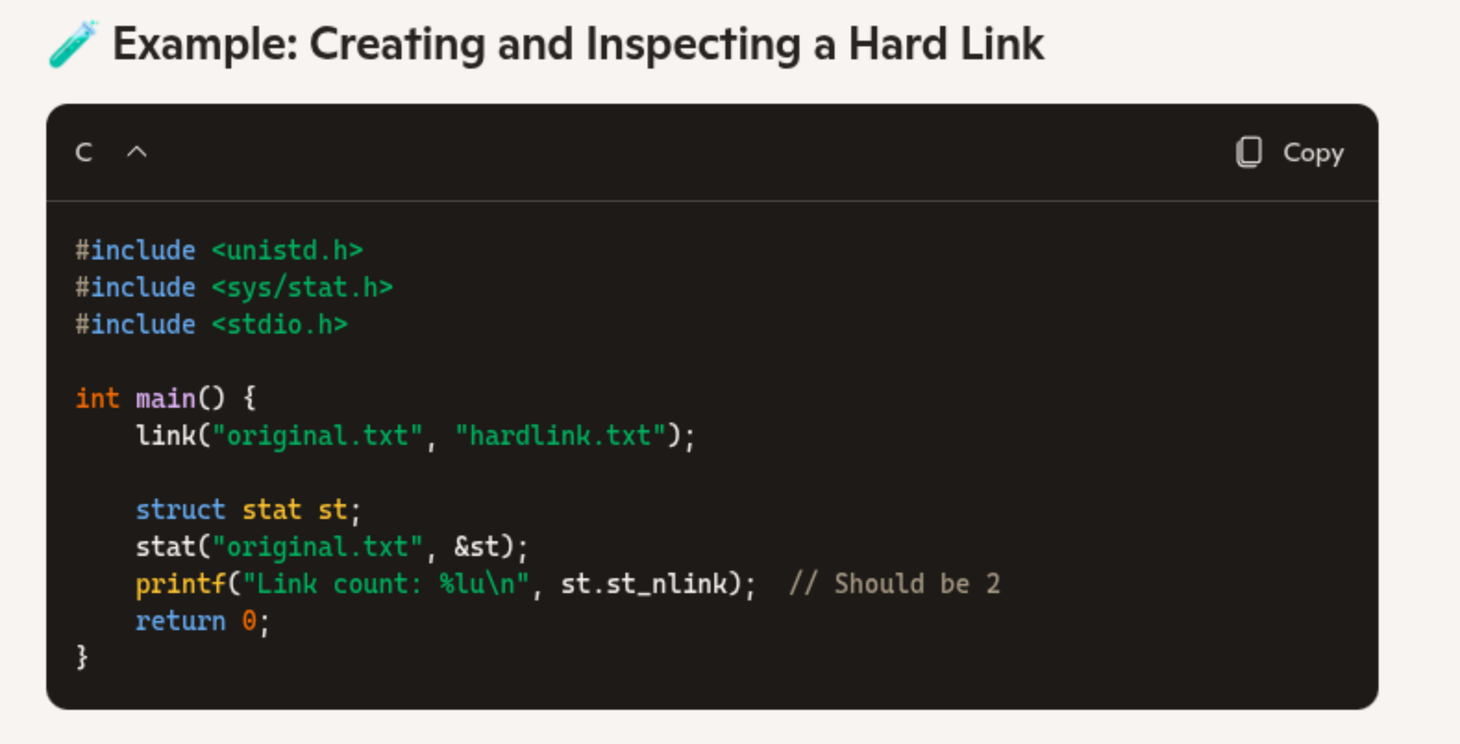


4. `unlink()`

- **Purpose**: Removes a directory entry (i.e., a hard link).
- **Prototype**:

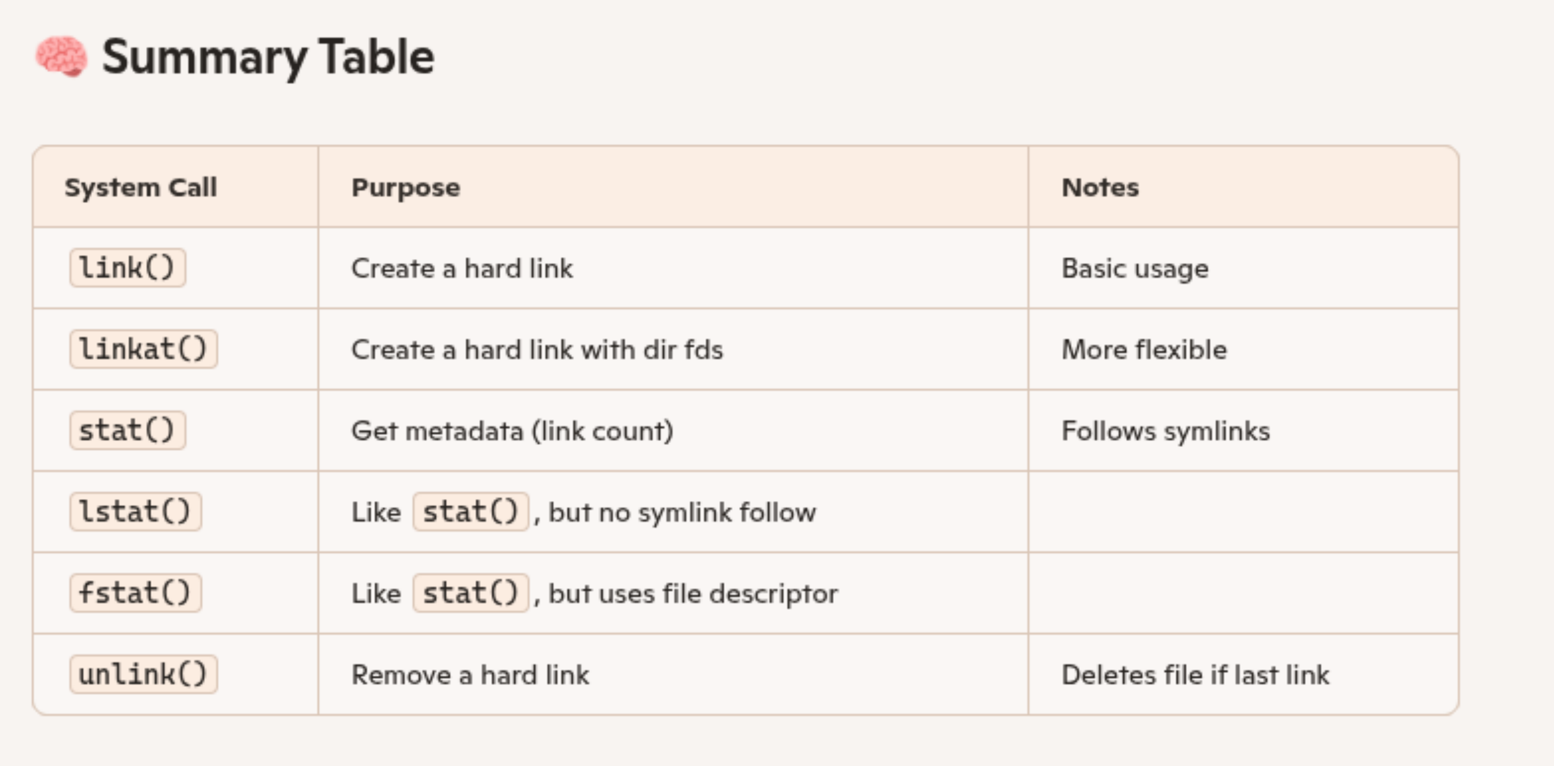
```c
int unlink(const char *pathname);
```

- **Behavior**: Decrements the link count. The file is deleted only when the count reaches zero and no process has it open.



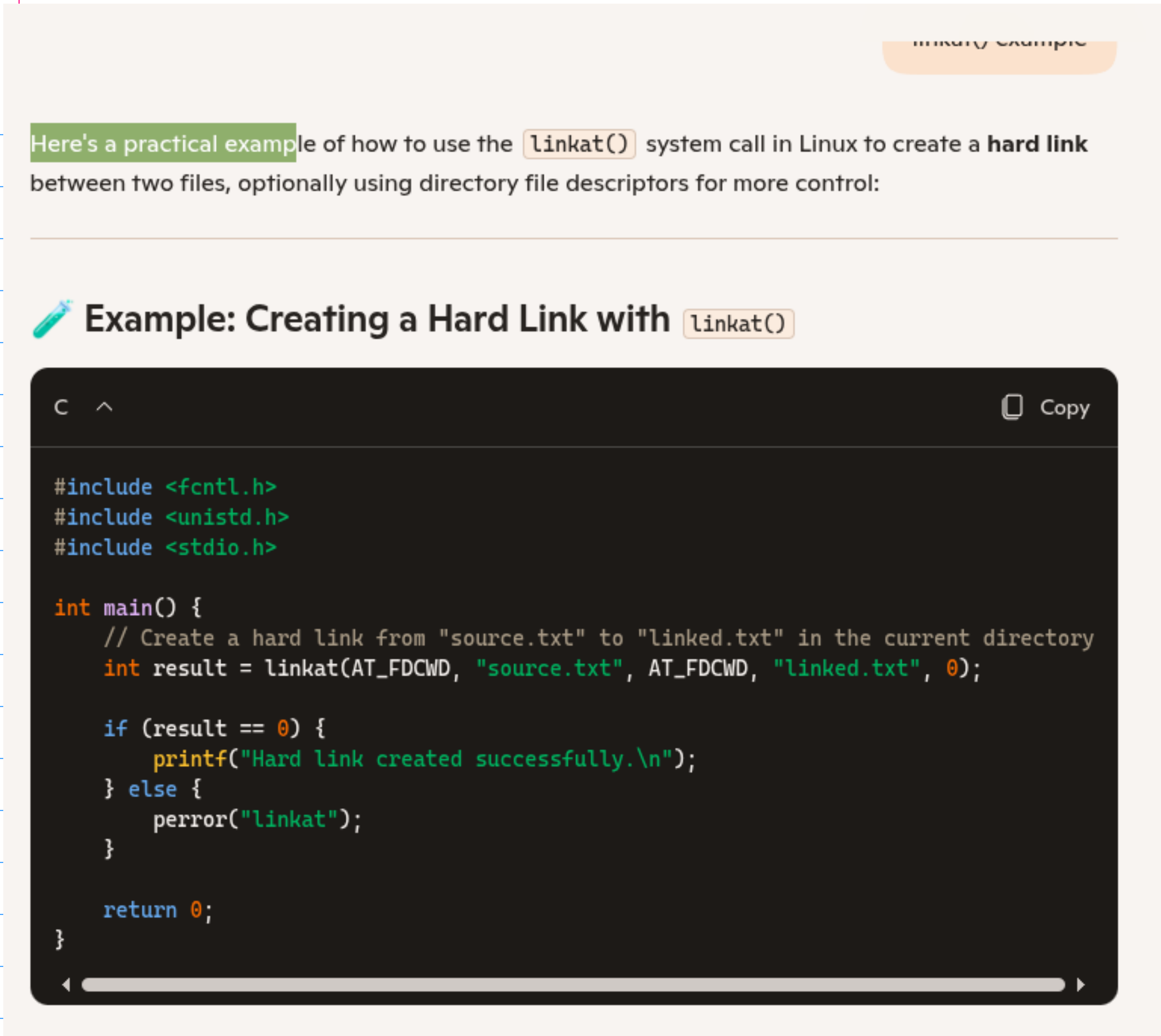
🧪 Example: Creating and Inspecting a Hard Link

```c
#include <unistd.h>
#include <sys/stat.h>
#include <stdio.h>

int main() {
    link("original.txt", "hardlink.txt");

    struct stat st;
    stat("original.txt", &st);
    printf("Link count: %lu\n", st.st_nlink);  // Should be 2
    return 0;
}
```



🧠 Summary Table

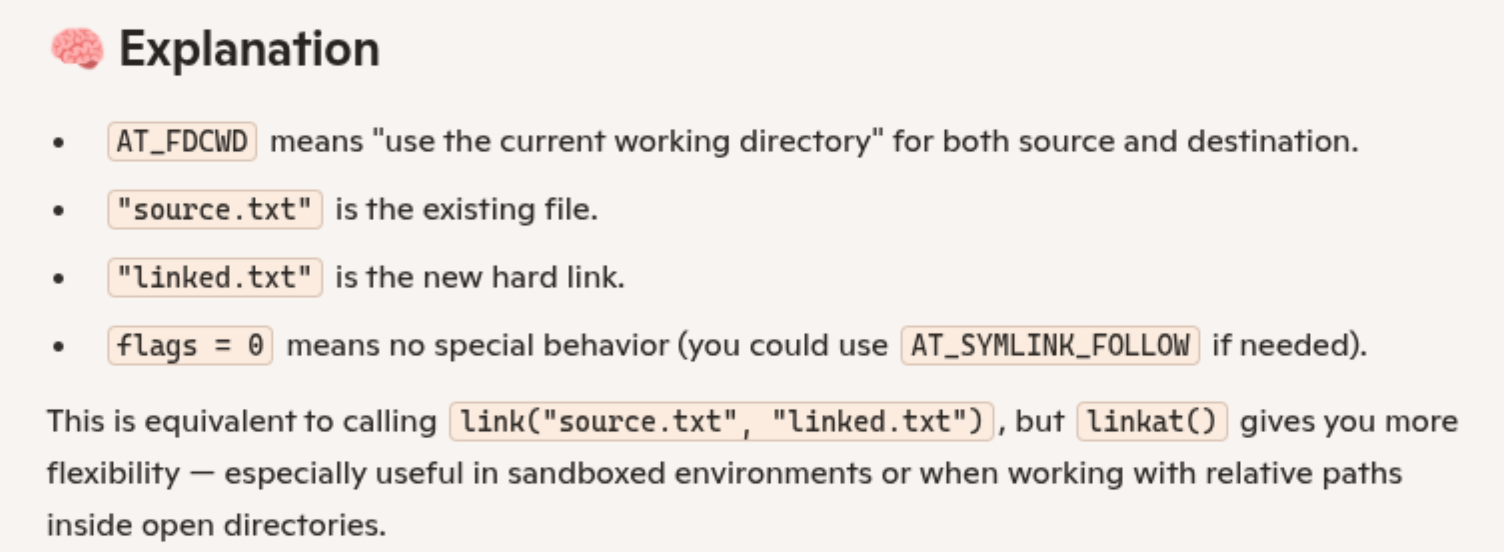
System Call	Purpose	Notes
`link()`	Create a hard link	Basic usage
`linkat()`	Create a hard link with dir fds	More flexible
`stat()`	Get metadata (link count)	Follows symlinks
`lstat()`	Like `stat()`, but no symlink follow	
`fstat()`	Like `stat()`, but uses file descriptor	
`unlink()`	Remove a hard link	Deletes file if last link



Here's a practical example of how to use the `linkat()` system call in Linux to create a **hard link** between two files, optionally using directory file descriptors for more control:

🧪 Example: Creating a Hard Link with `linkat()`

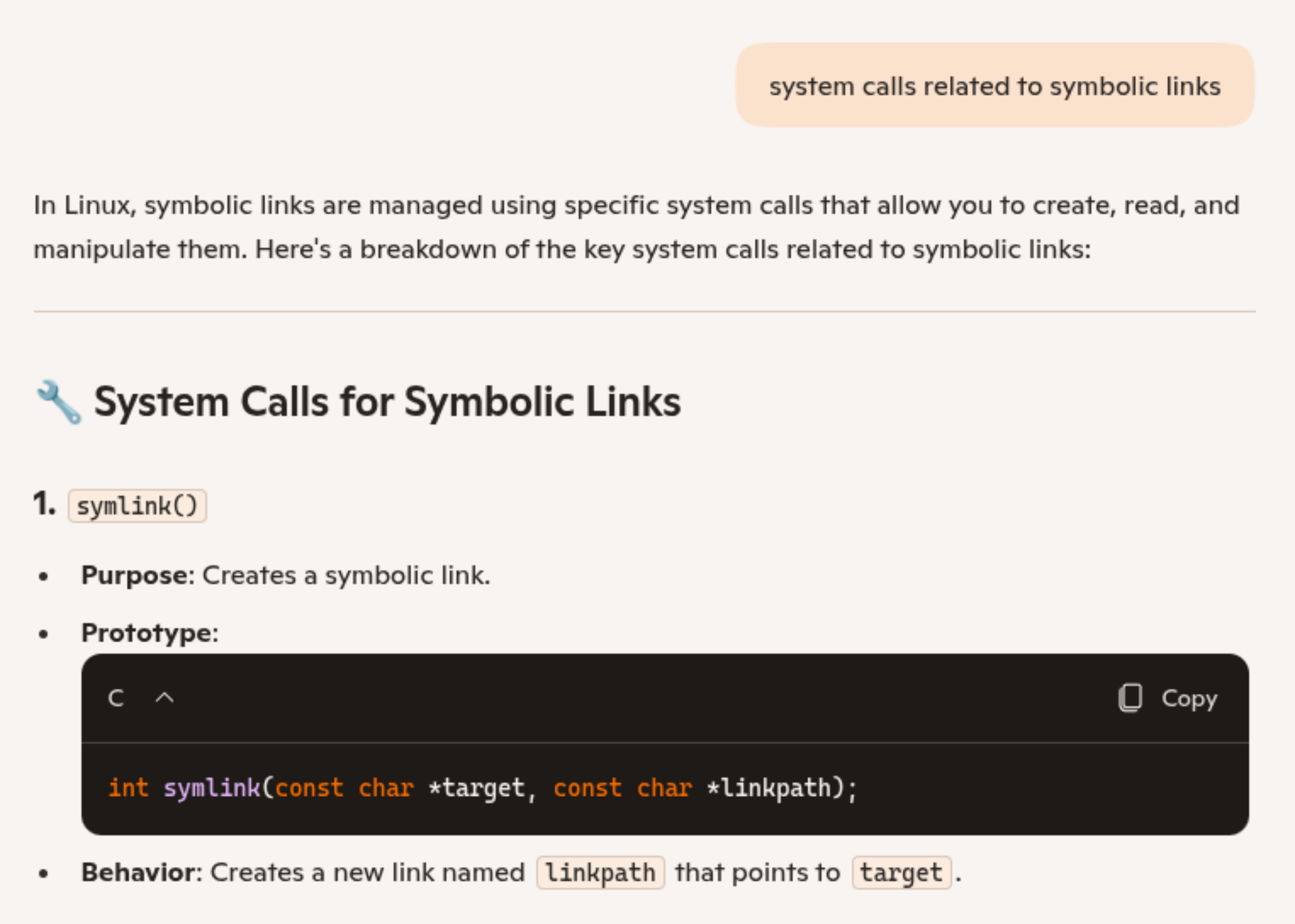
```c
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

int main() {
    // Create a hard link from "source.txt" to "linked.txt" in the current directory
    int result = linkat(AT_FDCWD, "source.txt", AT_FDCWD, "linked.txt", 0);

    if (result == 0) {
        printf("Hard link created successfully.\n");
    } else {
        perror("linkat");
    }

    return 0;
}
```



🧠 Explanation

- `AT_FDCWD` means "use the current working directory" for both source and destination.
- `"source.txt"` is the existing file.
- `"linked.txt"` is the new hard link.
- `flags = 0` means no special behavior (you could use `AT_SYMLINK_FOLLOW` if needed).

This is equivalent to calling `link("source.txt", "linked.txt")`, but `linkat()` gives you more flexibility — especially useful in sandboxed environments or when working with relative paths inside open directories.



system calls related to symbolic links

In Linux, symbolic links are managed using specific system calls that allow you to create, read, and manipulate them. Here's a breakdown of the key system calls related to symbolic links:

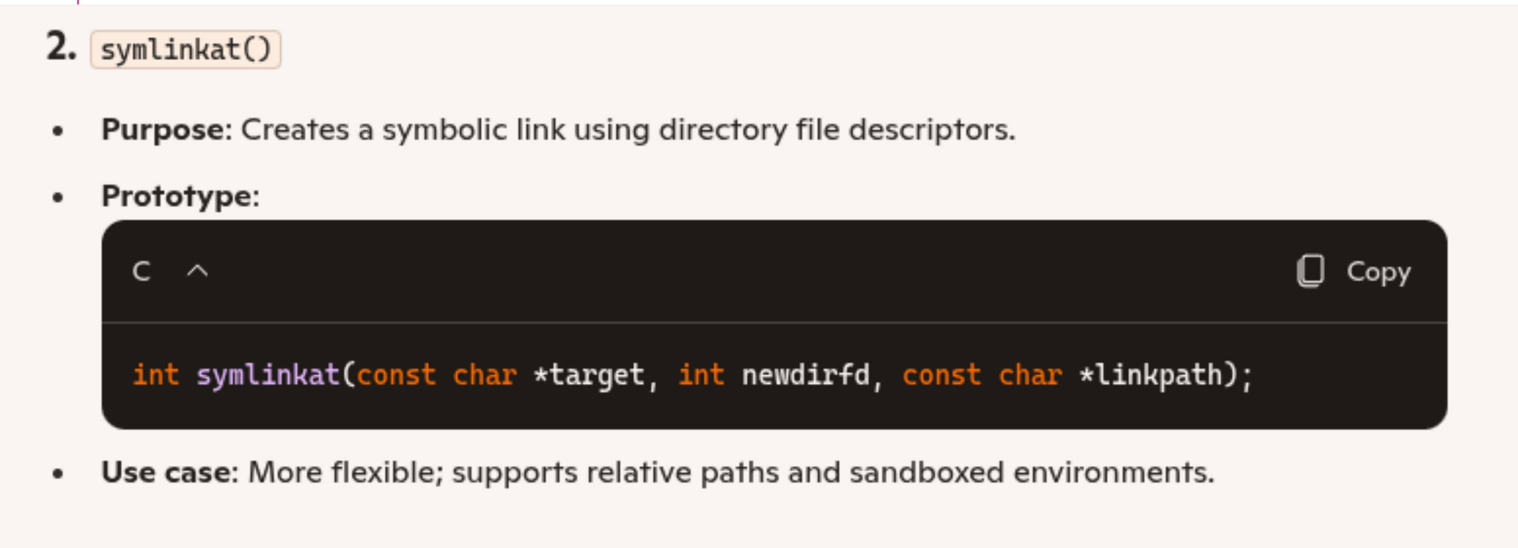
🔧 System Calls for Symbolic Links

1. `symlink()`

- **Purpose**: Creates a symbolic link.
- **Prototype**:

```c
int symlink(const char *target, const char *linkpath);
```

- **Behavior**: Creates a new link named `linkpath` that points to `target`.

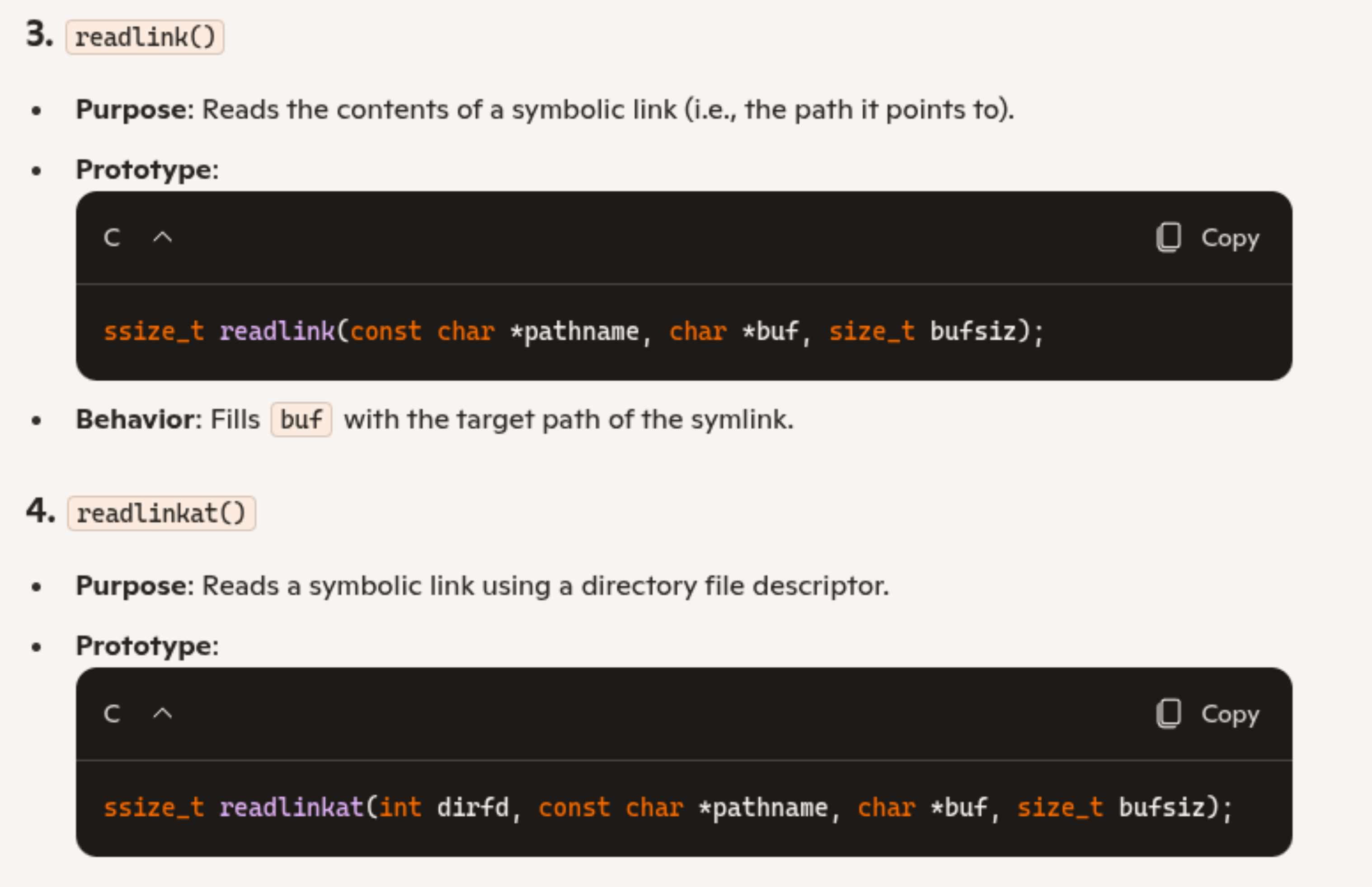


2. `symlinkat()`

- **Purpose**: Creates a symbolic link using directory file descriptors.
- **Prototype**:

```c
int symlinkat(const char *target, int newdirfd, const char *linkpath);
```

- **Use case**: More flexible; supports relative paths and sandboxed environments.



3. `readlink()`

- **Purpose:** Reads the contents of a symbolic link (i.e., the path it points to).
- **Prototype:**

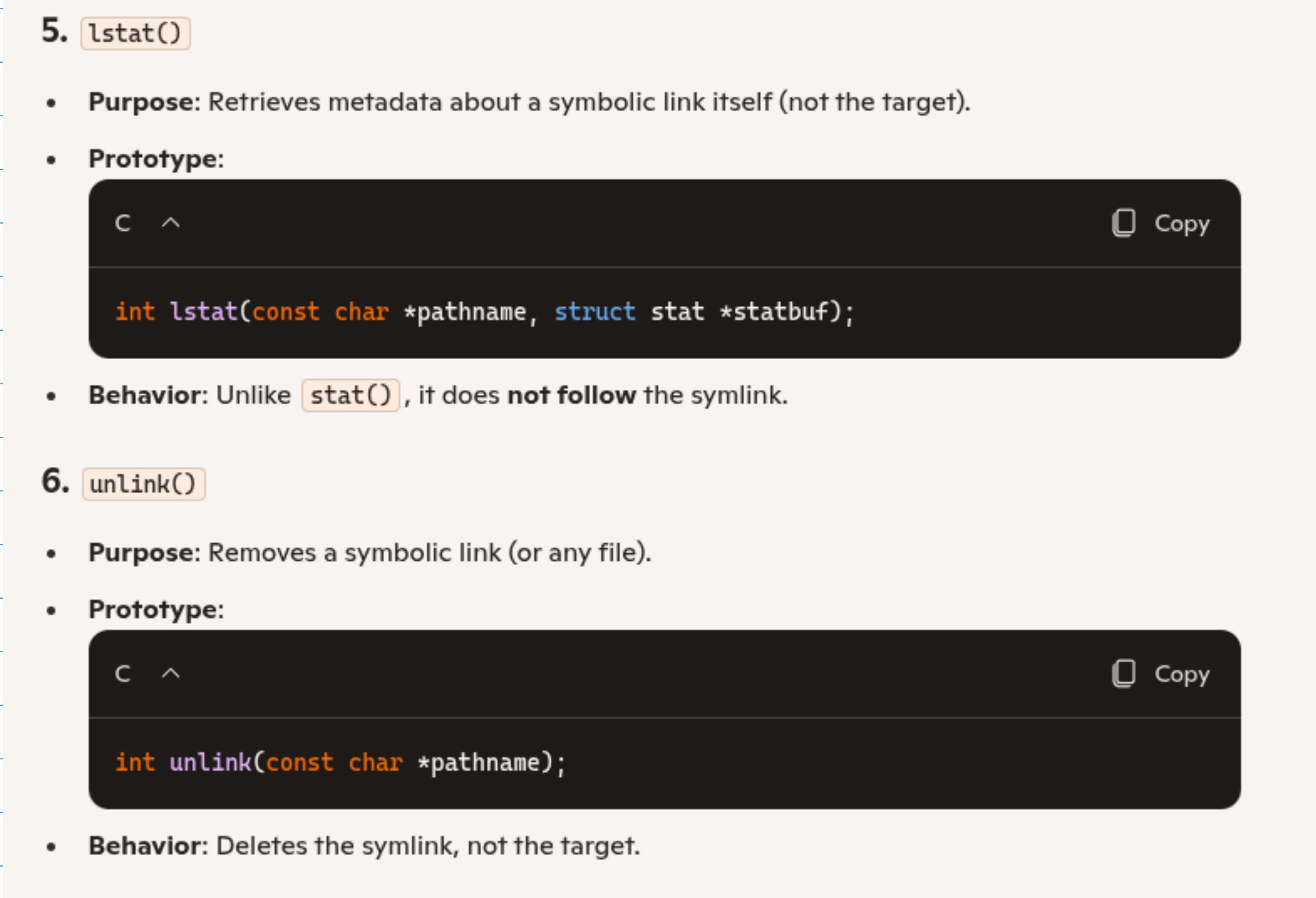
```c
ssize_t readlink(const char *pathname, char *buf, size_t bufsiz);
```

- **Behavior:** Fills `buf` with the target path of the symlink.

4. `readlinkat()`

- **Purpose:** Reads a symbolic link using a directory file descriptor.
- **Prototype:**

```c
ssize_t readlinkat(int dirfd, const char *pathname, char *buf, size_t bufsiz);
```



5. `lstat()`

- **Purpose**: Retrieves metadata about a symbolic link itself (not the target).
- **Prototype**:

```c
int lstat(const char *pathname, struct stat *statbuf);
```

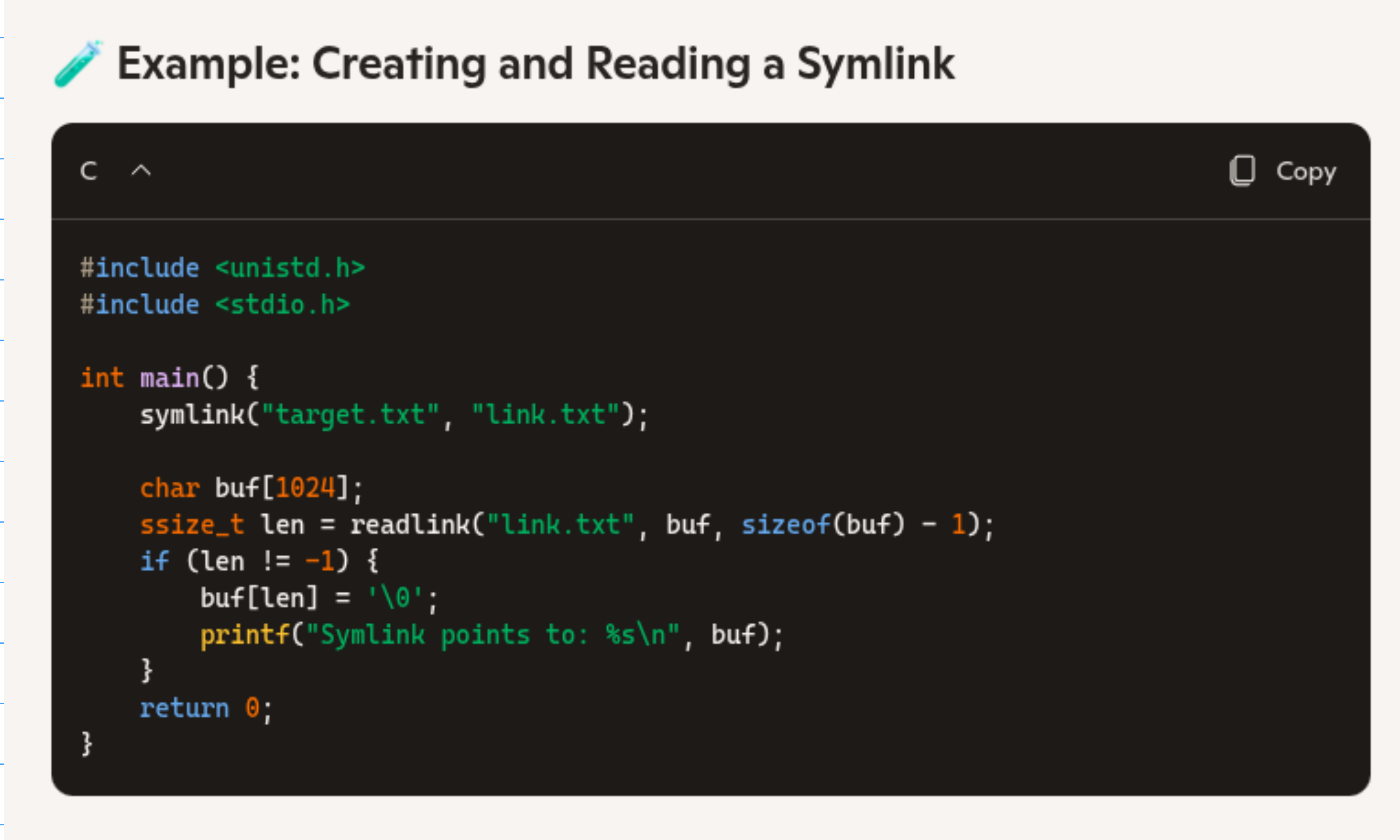
- **Behavior**: Unlike `stat()`, it does **not follow** the symlink.

6. `unlink()`

- **Purpose**: Removes a symbolic link (or any file).
- **Prototype**:

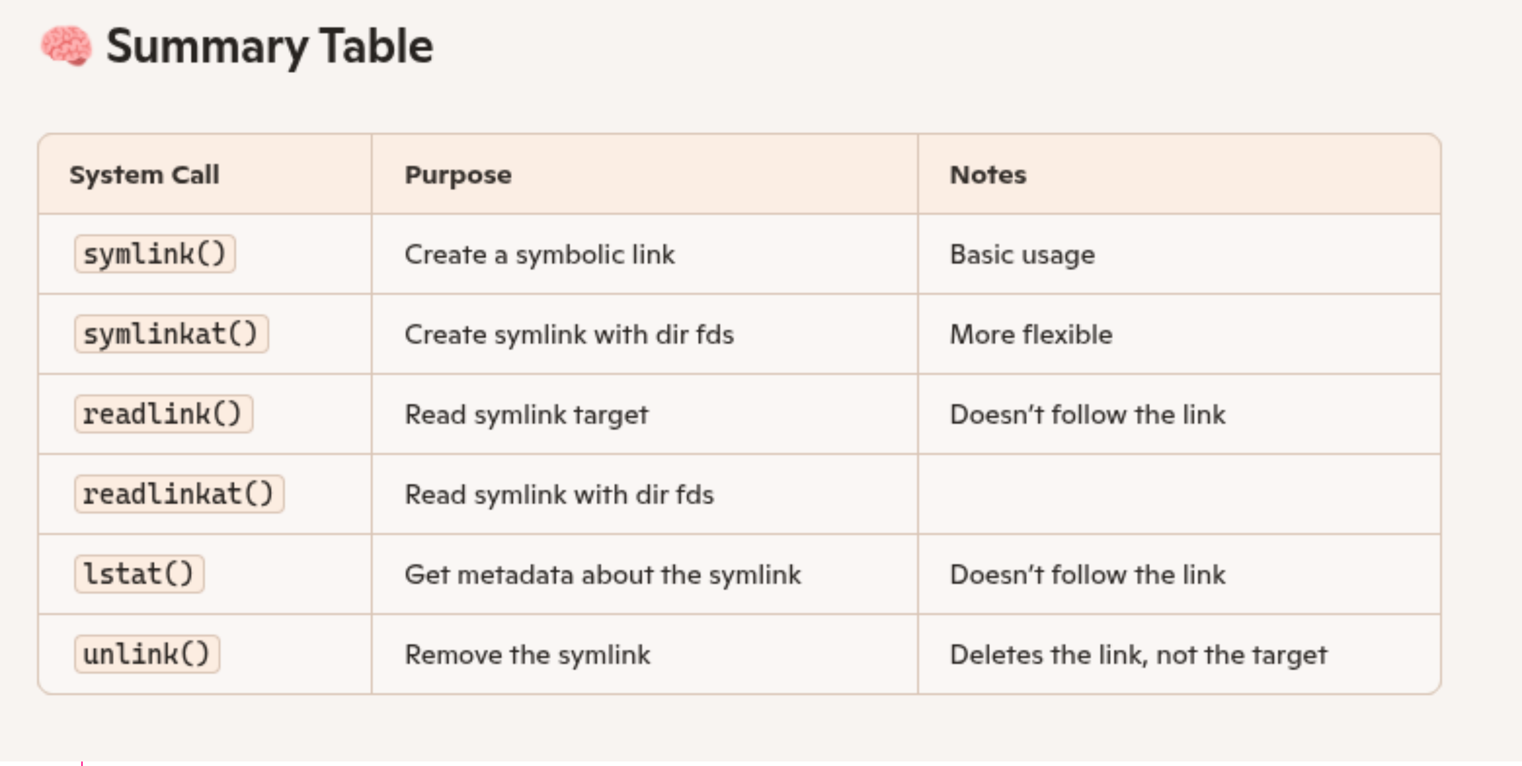
```c
int unlink(const char *pathname);
```

- **Behavior**: Deletes the symlink, not the target.



🧪 Example: Creating and Reading a Symlink

```c
#include <unistd.h>
#include <stdio.h>

int main() {
    symlink("target.txt", "link.txt");

    char buf[1024];
    ssize_t len = readlink("link.txt", buf, sizeof(buf) - 1);
    if (len != -1) {
        buf[len] = '\0';
        printf("Symlink points to: %s\n", buf);
    }
    return 0;
}
```



🧠 Summary Table

System Call	Purpose	Notes
`symlink()`	Create a symbolic link	Basic usage
`symlinkat()`	Create symlink with dir fds	More flexible
`readlink()`	Read symlink target	Doesn't follow the link
`readlinkat()`	Read symlink with dir fds	
`lstat()`	Get metadata about the symlink	Doesn't follow the link
`unlink()`	Remove the symlink	Deletes the link, not the target